מכון ויצמן למדע

WEIZMANN INSTITUTE OF SCIENCE

Thesis for the Degree
Doctor of Philosophy

by

Nir Piterman

Verification of Infinite-State Systems

Advisor
Prof. Amir Pnueli
Faculty of Mathematics and Computer Science
The Weizmann Institute of Science

October 6, 2004

Submitted to the Scientific Council of the
the Weizmann Institute of Science
Rehovot, Israel

Acknowledgments

I would like to express my deep gratitude to my advisor Prof. Amir Pnueli for giving me the freedom to choose my trail. Whenever I worked with Amir, he always amazed me with his wisdom and deep insights.

I am indebted to my other two advisors. First to Prof. Moshe Vardi, who brought me to verification and to automata theory in the first place. I could always count on Moshe for getting me out of tight places. Even in cases where he was not completely intimate with my problems, his large scale view and 'hawk eye' promised an advice as to where to go in order to progress. To Prof. Orna Kupferman, who was always there for me, helped me think about almost every result in this thesis, and accompanied me during these four years. I am also thankful for her friendship, good humor, and pleasant company.

I would like to thank Dr. Yonit Kesten, my roommate for the past 4 years. Yonit has always had the time to give me advice and share her knowledge with me. My partnership with Yonit culminated in a joint paper.

Finally, for my wife Jasmin for supporting me and helping me. Jasmin has always been there for me whenever I needed her. She lent me her ears whenever I needed to make important (and not important) decisions. For being my best friend and for helping me produce (or rather I helped her) the best result of the PhD: our daughter Shirley.

I dedicate this thesis to my daughter Shirley, who provided me with many joyous moments. Whenever I needed to clean my head from all this verification stuff she was ready to give me plenty of other things to do. Without the joy of having her I am not sure I would have completed this task.

Abstract

This thesis focuses on two verification problems, both handling infinite-state systems. The first is model checking of sequential infinite-state systems. We present an algorithm for model-checking linear-time properties of pushdown and prefix-recognizable systems. We then show that model-checking pushdown systems with regular labeling is equivalent to model-checking prefix-recognizable systems. We extend the automata-theoretic approach to infinite-state systems to handle global model checking for both linear-time and branching-time properties. We then use the methods developed for reasoning about infinite-state sequential systems to offer a solution to the emptiness problem of pushdown nondeterministic parity tree automata. Using this new solution, we consider model checking of non-regular specifications with respect to finite-state systems. We show that model checking this type of specifications with respect to pushdown systems is undecidable. We introduce the class of micro-macro stack graphs, which extends the class of prefix-recognizable graphs, and give elementary algorithms for model-checking linear-time and branching-time specifications with respect to micro-macro stack graphs.

The second problem is uniform verification of parameterized systems. We suggest a heuristic that enables proving the correctness of liveness properties for a large class of parameterized systems. We start with systems in which the connection between the processes is relatively simple and show how to extend the method to systems in which moves of processes may depend on the state of neighboring processes and even on all other processes. We exhibit the usefulness of our approach by proving liveness for a few protocols for mutual exclusion.

Contents

Chapter 1

Introduction

In formal verification we are interested in ensuring the correctness of designs. A first step is to formally define what correctness means. This is usually done by writing properties of the design in some formal specification language. Once a specification language has been chosen, depending on the type of design, we try to find algorithms or heuristics for verifying that the design meets its specification.

We usually distinguish between *algorithmic verification* and *deductive verification*. The first is fully automatic but restricted in the types of designs that it can handle (as well as in the size of designs). The second, requires much user guidance and interaction and is not guaranteed to terminate. However, it is applicable to a much wider range of designs. The heavy burden on the user of deductive verification is the reason why this method is not used more widely.

A major breakthrough has been the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [QS81, LP85, CES86, VW86a]. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior (for a survey, see [CGP99]). Model checking has two major advantages: it is fully automatic, and it produces, in the case of failure, a counterexample (an erroneous execution of the system). The introduction of symbolic model checking based on BDDs [Bry86, McM93] and bounded model checking based on satisfiability solvers [BCC$^+$99] has enabled model checking of very large state spaces. The increased capacity and the great ease of use of fully algorithmic methods, led to the acceptance of temporal model checking in hardware industry [BBDEL96, BLM01, CFF$^+$01].

Although the algorithmic aspects of temporal model checking over finite-state systems are well understood, many times, the sheer size of designs make current computing resources inadequate for model checking. The problem is aggravated when the design has an infinite number of states or when we want to consider families of designs that have an unbounded number of members. For example, in many cases we would like our design to relate to values with unbounded precision as in timed and hybrid systems. Designs may have unbounded memory such as a pushdown or a stack. In the context of networks (or out-of-order execution units), each participant may be finite, but we may wish to consider unbounded number of participants.

In this thesis we consider verification of two types of infinite-state systems. The first, systems that use a stack as a memory device. Thus, the amount of memory used by the design is unbounded, producing an infinite number of states. We develop algorithms for reasoning about temporal prop-

erties of this type of systems. The second, parameterized systems, where we would like to ensure that, for every number of copies of the system, the design still performs correctly. As reasoning about parameterized system is generally undecidable, we develop a heuristic that enables proving the correctness of liveness properties for a large class of parameterized systems. We have restricted the scope of this thesis to these two subjects. The results in [KPP03, KPV04], which relate to fair simulation relations, are not included here.

1.1 Systems with Pushdown / Stack Component

In recent years, an active thrust of research is the application of model checking to *infinite-state sequential rewrite systems*. These are systems in which a state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The configurations of the systems are described by words over some finite alphabet and the transitions by rewrite rules. The origin of this thrust is the important result by Müller and Schupp that the monadic second-order theory of *context-free graphs* is decidable [MS85]. Researchers then sought decidability results that relate to larger classes of systems and to simpler logics. This started with Burkart and Steffen, who developed an exponential-time algorithm for model-checking formulas in the *alternation-free* μ-calculus with respect to context-free graphs [BS92]. Researchers then went on to extend this result to the μ-calculus, on one hand, and to more general graphs on the other hand, such as *pushdown graphs* [BS95, Wal96], *regular graphs* [BQ96], and *prefix-recognizable graphs* [Cau96]. One of the most powerful results so far is an exponential-time algorithm by Burkart for model-checking formulas of the μ-calculus with respect to prefix-recognizable graphs [Bur97b]. See also [BS95, Cau96, BE96, Bur97a, FWW97, BS99, BCMS00]. Some of this theory has also been reduced to practice. Pushdown model checkers such as Mops [CW02], Moped [ES01, Sch02], and Bebop [BR00] (to name a few) have been developed. Successful applications of these model checkers to the verification of software are reported, for example, in [BR01, CW02].

In [KV00a], Kupferman and Vardi develop an automata-theoretic framework for reasoning about infinite-state sequential systems. The automata-theoretic approach to verification uses the theory of automata as a unifying paradigm for program specification, verification, and synthesis [WVS83, EJ91, Kur94, VW94, KVW00]. Automata enable the separation of the logical and the algorithmic aspects of reasoning about systems, yielding clean and asymptotically optimal algorithms.

The automata-theoretic framework for reasoning about finite-state systems has proven to be very versatile. Automata are the key to techniques such as on-the-fly verification [GPVW95], and they are useful also for modular verification [KV98], partial-order verification [GW94, WW96], verification of real-time and hybrid systems [HKV96, DW99], verification of open systems [AHK97, KV99], and verification of pushdown and prefix-recognizable graphs [KV00a]. Many decision and synthesis problems have automata-based solutions and no other solution for them is known [EJ88, PR89, KV00b]. Automata-based methods have been implemented in industrial automated-verification tools (c.f., COSPAN [HHK96] and SPIN [Hol97, VB00]).

The automata-theoretic framework for reasoning about infinite-state sequential systems is based on the observation that states of such systems can be viewed as nodes in an infinite tree, and transitions between states can be simulated by finite-state automata. Checking that the system satisfies a branching temporal property can then be done by an alternating two-way automaton. The two-way alternating automaton starts checking the input tree from the root. It then spawns several copies of itself that may go in different directions in the tree. Each new copy can spawn other new copies and so on. The automaton accepts the input tree if all spawned copies agree on acceptance.

Thus, copies of the alternating automaton navigate through the tree and check the branching temporal property. The method in [KV00a] handles prefix-recognizable systems, and properties specified in the μ-calculus. The method appears to be, like the automata-theoretic framework for finite-state systems, very versatile, and it has further applications: the μ-calculus model-checking algorithm can be easily extended to graphs with *regular labeling* (that is, graphs in which each atomic proposition p has a regular expression describing the set of states in which p holds) and *regular fairness constraints*, to μ-calculus with *backward modalities*, and to checking *realizability* of μ-calculus formulas with respect to infinite-state sequential environments. All the above are achieved using a reduction to the emptiness problem for alternating two-way tree automata where the location of the alternating automaton on the infinite tree indicates the contents of the pushdown store.

1.1.1 Model-Checking Linear-Time Properties

Our first contribution, presented in Chapter 2, is an extension of the the automata-theoretic framework developed by Kupferman and Vardi to handle also specifications in linear temporal logic. Model-checking linear-time specifications with respect to pushdown graphs was considered in [BEM97] and improved in [EHRS00, EKS01]. The model-checking algorithm is exponential in the formula, but is only polynomial in the system [BEM97].

We note that the μ-calculus is sufficiently strong to express all properties expressible in the linear temporal logic LTL (and in fact, all properties expressible by an ω-regular language) [Dam94]. Thus, by translating LTL formulas into μ-calculus formulas we can use the solution in [KV00a] for μ-calculus model checking in order to solve LTL model checking. This solution, however, is not optimal. This has to do both with the fact that the translation of LTL to μ-calculus is exponential, as well as the fact that the solution for μ-calculus model checking is based on tree automata. A tree automaton splits into several copies when it runs on a tree. While splitting is essential for reasoning about branching properties, it has a computational price. For linear properties, it is sufficient to follow a single computation of the system, and tree automata seem too strong for this task. For example, the application of the framework developed in [KV00a] to pushdown systems and LTL properties results in an algorithm that is doubly-exponential in the formula and exponential in the system, which is not optimal [BEM97].

In order to handle model checking of linear-time properties, we introduce *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree. In particular, *two-way* nondeterministic path automata enable exactly the type of navigation that is required in order to check linear properties of infinite-state sequential systems. We study the expressive power and the complexity of the decision problems for (two way) path automata. The fact that path automata follow a single path in the tree makes them very similar to two-way nondeterministic automata on infinite words. This enables us to reduce the membership problem (whether an automaton accepts the tree obtained by unwinding a given finite labeled graph) of two-way nondeterministic path automata to the emptiness problem of one-way alternating Büchi automata on infinite words, which was studied in [VW86b]. This leads to a quadratic upper bound for the membership problem for two-way nondeterministic path automata.

Using path automata we are able to solve the problem of LTL model checking with respect to pushdown and prefix-recognizable systems by a reduction to the membership problem of two-way nondeterministic path automata. Our automata-theoretic technique matches the known upper

bound for model-checking LTL properties on pushdown systems [BEM97, EHRS00]. In addition, the automata-theoretic approach provides the first solution for the case where the system is prefix-recognizable.

Usually, the labeling of the state depends on the internal state of the system and the top of the store. Our framework also handles *regular labeling*, where the label depends on whether the word on the store is a member in some regular language. The complexity is exponential in the nondeterministic automata that describe the labeling, matching the known bound for pushdown systems and linear-time specifications [EKS01]. The automata-theoretic techniques for handling regular labeling and for handling the regular transitions of a prefix-recognizable system are very similar. This leads us to the understanding that regular labeling and prefix-recognizability have exactly the same power.

1.1.2 Global Model Checking

We usually distinguish between *local* and *global* model checking. In the first setting we are given a specific state of the system and determine whether it satisfies a given property. In the second setting we compute (a finite representation of) the set of states that satisfy a given property. For many years global model-checking algorithms were the standard; in particular, CTL model checkers [CES86], and symbolic model checkers [BCM+92] perform global model checking. While local model checking holds the promise of reduced computational complexity [SW91] and is more natural for explicit LTL model checking [CVWY92], global model checking is especially important in cases where model checking is only part of the verification process. For example, in [CKV01, CKKV01], global model checking is used to supply coverage information, which informs us what parts of the design under verification are relevant to the specified properties. In [Sha00, LBBO01], an infinite-state system is abstracted into a finite-state system. Global model checking is performed over the finite-state system and the result is then used to compute invariants for the infinite-state system. In [PRZ01], results of global model checking over small instances of a parameterized system are generalized to invariants for every value of the system's parameter.

Traditionally, automata-theoretic techniques provide algorithms only for local model checking [CVWY92]. In particular, the framework of [KV00a] and its extension mentioned above solve only local model checking. As model checking in the automata-theoretic approach is reduced to the emptiness of an automaton, it seems that this limitation to local model checking is inherent to the approach. For finite-state systems we can reduce global model checking to local model-checking by iterating over all the states of the system, which is essentially what happens in symbolic model checking of LTL [BCM+92]. For infinite-state systems, however, such a reduction cannot be applied.

Our second contribution, presented in Chapter 3, is to remove this limitation of automata-theoretic techniques. We show that the automata-theoretic approach to infinite-state sequential systems generalizes nicely to global model checking. Thus, all the advantages of using automata-theoretic methods, e.g., the ability to handle regular labeling and regular fairness constraints, the ability to handle μ-calculus with backward modalities, and the ability to check realizability [KV00a, ATM03], apply also to the more general problem of global model checking. Our result matches the upper bounds for global model checking established in [BEM97, EHRS00, EKS01, KPV02a, Cac02]. Our contribution is in showing how this can be done uniformly in an automata-theoretic framework rather than via an eclectic collection of techniques.

1.1.3 Pushdown Specifications

We then proceed to consider *infinite-state specifications*. Almost all existing work on model checking considers specification formalisms that define *regular* sets of words, trees, or graphs: formulas of LTL, μ-calculus, and even monadic second-order logic can all be translated to automata [Büc62, Rab69, EJ91], and in fact many model-checking algorithms (for both finite-state and infinite-state systems) first translate the given specification into an automaton and reason about the structure of this automaton (cf., [VW86a, BEM97, KV00a]). Sometimes, however, the desired behavior is non-regular and cannot be specified by a finite-state automaton. Consider for example the property "p is inevitable", for a proposition p. That is, in every computation of the system, p eventually holds. Clearly, this property is regular and is expressible as $\forall\Diamond p$ in both CTL [CES86] and LTL [Pnu77]. On the other hand, the property "p is uniformly inevitable", namely, there is some time i such that in every computation of the system, p holds at time i, is not expressible by a finite automaton on infinite trees [Eme87], and hence, it is non-regular. More examples to useful non-regular properties are given in [SCFG84], where the specification of unbounded message buffers is considered.

The need to specify non-regular behaviors led Bouajjani et al. [BER94, BEH95] to consider logics that are a combination of CTL and LTL with Presburger Arithmetic. The logics, called PCTL and PLTL, use variables that range over natural numbers. The variables are bound to the occurrences of state formulas and comparison between such variables is allowed. The non-regular property discussed above can be specified in PCTL and PLTL. For example, we can specify uniform inevitability in PCTL as $\widetilde{\exists} i \;.\; \forall[x : \mathbf{true}](x = i \rightarrow p)$, where the $\widetilde{\exists}$ quantifier quantifies over natural numbers, the $\forall$ quantifier quantifies over computations of the system, and the combinator $[x : \mathbf{true}]$ binds the variable x to count the number of occurrences of the state formula **true**. Bouajjani et al. consider the model-checking problem for the logics PCTL and PLTL over finite-state (regular) systems and over infinite-state (non-regular) systems. The logics turned out to be too strong: the model checking of both PCTL and PLTL over finite-state systems is undecidable. They proceed to restrict the logics to fragments for which model checking over finite-state systems and context-free systems is decidable. The property "p is uniformly inevitable" is expressible in the restricted (decidable) fragments of PCTL and PLTL.

Uniform inevitability is clearly expressible by a *nondeterministic pushdown tree automaton*. Pushdown tree automata are finite-state automata augmented by a pushdown store. Like a nondeterministic finite-state tree automaton, a nondeterministic pushdown tree automaton starts reading a tree from the root. At each node of the tree, the pushdown automaton consults the transition relation and splits into independent copies of itself to each of the node's successors. Each copy has an independent pushdown store that diverges from the pushdown store of the parent. We then check what happens along every branch of the run tree and determine acceptance. In order to express uniform inevitability, the automaton guesses the time i, pushes i elements into the pushdown store, and, along every computation, pops one element with every move of the system. When the pushdown store becomes empty, the automaton requires p to hold. In [PI95], Peng and Iyer study more properties that are non-regular and propose to use nondeterministic pushdown tree automata as a strong specification formalism. The model studied by [PI95] is *empty store*: a run of the automaton is accepting if the automaton's pushdown store gets empty infinitely often along every branch in the run tree.

Recently, Alur et al. introduced the logic CARET [AEM04]. CARET is a linear temporal logic that can specify non-regular properties. Model-checking CARET with respect to pushdown systems is decidable in exponential time [AEM04]. We note that CARET is less expressive than

pushdown automata. The study of CARET inspired Alur et al. to introduce *visibly pushdown languages*, which are a subset of the context-free languages, for which model checking with respect to pushdown systems is decidable [AM04].

In Chapter 4 we study the model-checking problem for specifications given by nondeterministic pushdown tree automata. We consider both finite-state (regular) and infinite-state (non-regular) systems. We show that, for finite-state systems, the model-checking problem is solvable in time exponential in both the system and the specification, even for nondeterministic pushdown parity tree automata – a model that is stronger than the one studied in [PI95, AEM04]. On the other hand, the model-checking problem for context-free systems is undecidable – already for a weak type of pushdown tree automata.

1.1.4 Beyond Prefix-Recognizable

It is well known that the class of prefix-recognizable graphs strictly contains the class of pushdown graphs [Cau96]. More powerful notion of rewrite rules yield even larger classes of graphs. In *rational* graphs the transition relation is described by a two-headed finite automaton. We allow transitions between two configurations (words) that are accepted by the two-headed automaton. In *synchronized rational* graphs we require in addition that the two-headed automaton move on the two words synchronously. The class of synchronized rational graphs strictly contains the class of prefix-recognizable graphs and in turn is strictly contained in the class of rational graphs. Only the first-order theory of synchronized rational graphs is, however, decidable (cf. [Büc60, Tho01]). It is undecidable even to determine if some vertex is reachable from another vertex (cf. [Tho01]). For rational graphs even first-order theory is undecidable [Mor00]. There are also larger classes of systems for which monadic second-order theory is decidable. Walukiewicz [CW98, Wal02] shows that decidability of monadic second-order theory is preserved under unfolding. Caucal then shows that by iterating unfolding and prefix-rewriting we get larger classes of systems while preserving the decidability of monadic second-order theory [Cau02]. These results, however, handle only the monadic second-order theory and do not include simpler logics for which decidability may be elementary.

In Chapter 5, we introduce the class of *micro-macro stack graphs*, which strictly contains the class of prefix-recognizable graphs and for which model-checking μ-calculus formulas is decidable in elementary time. Every graph in our class has a simple finite representation in terms of natural rewrite rules. The extension from prefix-recognizable graphs to micro-macro stack graphs is analogous to the extension from pushdown graphs to prefix-recognizable graphs.

Micro-macro stack graphs are the configuration graphs of stack automata [GGH67b, GGH67a, HU79]. Like pushdown automata, stack automata have a finite but unbounded store, they can change only the top of the store by either removing the letter on top of the store or by adding a finite sequence of letters on top of the store. Unlike pushdown automata, stack automata can read the entire contents of their store. A stack automaton can navigate on its store, checking its entire contents. It can change the contents of the store only when it visits the top of the store.

In our framework, states of the stack automaton are partitioned into micro and macro states. We refer to such stack automata as micro-macro stack automata. The nodes of a micro-macro stack graph correspond to configurations of a micro-macro stack automaton whose state is a macro state. Edges of the graph correspond to the change performed via a sequence of micro states.

We show that the class of micro-macro stack graphs strictly contains the class of prefix-

recognizable graphs. We give two examples of micro-macro stack graphs that are not prefix-recognizable. We extend the automata-theoretic approach to model-checking infinite-state systems to handle micro-macro stack graphs. We give an elementary time algorithms for both μ-calculus and LTL model checking.

Pushdown automata can be seen as the first level in an infinite hierarchy of high-order pushdown automata. A high-order pushdown automaton of level $i+1$, has a pushdown store in which every element is an i-order pushdown store. The automaton can do i-order actions on the top i-order element in its store. It can also do an $i+1$-order pop and remove the top i-order element, or an $i+1$-order push in which it copies the top i-order element and puts a fresh copy of this i-order element as the new top of store. Knapik et al. show that the monadic second-order theory of *high-order pushdown graphs*, the configuration graphs of high-order pushdown automata, is decidable [KNU03]. The class of micro-macro stack graphs is contained in the class of high-order pushdown graphs. A second-order pushdown automaton mimics a stack automaton in the following way. It goes into the stack by doing a second-order push (i.e., putting a fresh copy of the top first-order element on the store) and a first-order pop (i.e., removing the top letter of the first-order pushdown store). It moves towards the top of the stack by doing a second-order pop. It follows that the monadic second-order theory of micro-macro stack graphs is decidable. In parallel to the work presented here, Cachat solved the problem of μ-calculus model checking over high-order pushdown graphs [Cac03]. His solution for high-order pushdown graphs of level 2 (which include micro-macro stack graphs) has the same complexity as our own. His algorithm extends also to higher levels of high-order pushdown graphs. Carayol and Wöhrle later showed that the hierarchy of high-order pushdown automata is exactly equivalent to the hierarchy created by iterating the unfolding and prefix-rewriting of Walukiewicz and Caucal [CW03].

1.2 Parameterized Systems

The problem of *uniform verification of parameterized systems* is a very challenging problem in verification today. Given a parameterized system $S(N) : P[1] \parallel \cdots \parallel P[N]$ and a property p, uniform verification attempts to verify $S(N) \models p$ for every $N > 1$. Model checking is an excellent tool for *debugging* parameterized systems because, if the system fails to satisfy p, this failure can be observed for a specific (and usually small) value of N. However, once all the observable bugs have been removed, the question remains whether the system is correct for all $N > 1$.

The problem of uniform verification of parameterized systems is, in general, undecidable [AK86]. There are two possible remedies to this situation: either we should look for restricted families of parameterized systems for which the problem becomes decidable, or devise methods that are sound but, necessarily, incomplete, and hope that the system of interest yields to one of these methods.

There are many representatives for both approaches. Among the representatives of the first approach we can count the work of German and Sistla [GS92] that assumes a parameterized system where processes communicate synchronously, and shows how to verify single-index properties (i.e., properties that relate to a single process of the parameterized system). Similarly, Emerson and Namjoshi provide a decision procedure for proving a restricted set of properties on ring algorithms [EN95], and prove that verification of synchronously communicating processes is PSPACE-complete [EN96]. Many of these methods fail when we move to asynchronous systems where processes communicate by shared variables. Perhaps the most advanced of this approach is the paper [EK00] that considers a general parameterized system allowing several different classes of processes. However, this work provides separate algorithms for the cases that the guards are either all disjunctive or

all conjunctive. A protocol such as the cache example considered in [PRZ01] that contains some disjunctive and some conjunctive guards cannot be handled by the methods of [EK00].

The sound but incomplete methods include methods based on explicit induction [EN95] network invariants, which can be viewed as implicit induction [KM95, WL89, HLR92, LHR97], methods that can be viewed as abstraction and approximation of network invariants [BCG86, SG89, CGJ95, KP00], and other methods that can be viewed as based on abstraction [ID96]. In [CR99a, CR99b, CR00], the authors use structural induction based on the notion of a network invariant but significantly enhance its range of applicability by using a generalization of the data-independence approach that provides a powerful abstraction capability, allowing it to handle network with parameterized topologies. Most of these methods require the user to provide auxiliary constructs, such as a network invariant or an abstraction mapping. Other attempts to verify parameterized protocols such as Burn's protocol [JL98] and Szymanski's algorithm [GZ98, MAB+94] relied on abstraction functions or lemmas provided by the user. The work in [LS97] deals with the verification of safety properties of parameterized networks by abstracting the behavior of the system. The theorem prover PVS [SOR93] is used to discharge the generated verification conditions.

Among the automatic incomplete approaches, we should mention the methods relying on "regular model checking" [KMM+97, ABJN99, JN00, PS00], where linear configurations of processes (e.g., networks of linear or cyclic topology) are represented as a word in a regular language. This was extended to a word in a nonregular language [FP01] and to more complex structures such as trees [AJMd02, BT02]. Unfortunately, many of the systems analyzed by this method cause the analysis procedure to diverge and special *acceleration* procedures have to be applied that, again, requires user ingenuity and intervention.

The works in [ES96, ES97, CEFJ96, GS97] study symmetry reduction in order to deal with state explosion. The work in [ID96] detects symmetries by inspection of the system description.

1.2.1 Invisible Assertions

One method, which can always be applied to verify parameterized systems, is based on *deductive verification* [MP95]. For example, in order to verify that a parameterized system satisfies the invariance property $\Box p$, we may use rule INV of [MP95]: in order to prove that assertion r is an invariant of the program P, the rule requires coming up with an auxiliary assertion φ that is inductive (i.e., is implied by the initial condition and is preserved under every computation step) and that strengthens (implies) r. In rare cases, the property r is already inductive. In all other cases, the deductive verifier has to perform the following tasks:

T1. Divine (invent) the auxiliary assertion φ.
T2. Establish the inductiveness of φ and that φ implies r.

Performing interactive first-order verification of implications such as the premises above for a nontrivial system is never an easy task. Neither is it a one-time task, since the process of developing the auxiliary invariants requires iterative verification trials, where failed efforts lead to correction of the previous candidate assertion into a new candidate.

The papers [PRZ01, APR+01] introduce the method of invisible invariants. The method offers a procedure for the automatic generation of the auxiliary assertion φ for parameterized systems, as well as an efficient algorithm for checking the validity of the premises of INV.

The generation of invisible auxiliary constructs is based on the following idea: it is often the case that an auxiliary assertion φ for a parameterized system $S(N)$ has the form $\forall i : [1..N].q(i)$ or, more

generally, $\forall i \neq j.q(i,j)$. We construct an instance of the parameterized system taking a fixed value N_0 for the parameter N. For the finite-state instantiation $S(N_0)$, we compute, using BDDs, some assertion ψ that we wish to generalize to an assertion in the required form. Let r_1 be the projection of ψ on process $P[1]$, obtained by discarding references to variables that are local to all processes other than $P[1]$. We take $q(i)$ to be the generalization of r_1 obtained by replacing each reference to a local variable $P[1].x$ by a reference to $P[i].x$. The obtained $q(i)$ is our candidate for the body of the inductive assertion $\varphi : \forall i.q(i)$. We refer to this generalization procedure as *project&generalize*. For example, when computing invisible invariants, ψ is the set of reachable states of $S(N_0)$. The procedure can be easily generalized to generate assertions of the type $\forall i_1, \ldots, i_k.p(\vec{i})$.

Having obtained a candidate for the assertion φ, we still have to check the validity of the premises of the proof rule we wish to employ. Under the assumption that our assertional language is restricted to the predicates of equality and inequality between bounded-range integer variables (which is adequate for many of the parameterized systems we considered), [PRZ01] proves a *small model* theorem, according to which, for a certain type of assertions, there exists a (small) bound N_0 such that such an assertion is valid for every N iff it is valid for all $N \leq N_0$. This enables using BDD-techniques to check the validity of such an assertion. The cases covered by the theorem are those whose premises can be written in the form $\forall\vec{i}\exists\vec{j}.\psi(\vec{i},\vec{j})$, where $\psi(\vec{i},\vec{j})$ is a quantifier-free assertion that may refer only to the global variables and the local variables of $P[i]$ and $P[j]$ ($\forall\exists$-*assertions* for short).

Being able to validate the premises on $S[N_0]$ has the additional important advantage that the user never sees the automatically generated auxiliary assertion φ. This assertion is produced as part of the procedure and is immediately consumed in order to validate the premises of the rule. Being generated by symbolic BDD-techniques, the representation of the auxiliary assertions is often extremely unreadable and non-intuitive, and it usually does not contribute to a better understanding of the program or its proof. Because the user never gets to see it, this method was named the "method of *invisible invariants*."

As shown in [PRZ01, APR+01], embedding a $\forall\vec{i}.q(\vec{i})$ candidate inductive invariant in INV results in premises that fall under the small-model theorem. In Chapter 6, we extend the method of invisible invariants to apply to proofs of the second most important class of properties – the class of *response properties*. Response properties are liveness properties that can be specified by the temporal formula $\Box(q \rightarrow \Diamond r)$ and guarantees that every q-state is eventually followed by an r-state. To handle response properties, we consider a certain variant of rule WELL [MP91], which establishes the validity of response properties under the assumption of *justice* (weak fairness). As is well known to users of this and similar rules, such a proof requires the generation of two kinds of auxiliary constructs: *helpful assertions* h_i, which characterize, for transition τ_i, the states from which the transition is helpful in promoting progress towards the goal (r), and *ranking functions*, which measure progress towards the goal.

In order to apply *project&generalize* to the automatic generation of the ranking functions, we propose a variant of rule WELL. In this variant rule, called DISTRANK, we associate, with each potentially helpful transition τ_i, an individual ranking function $\delta_i : \Sigma \mapsto [0..c]$, mapping states to integers in a small range $[0..c]$ for some fixed small constant c. The global ranking function can be obtained by forming the multi-set $\{\delta_i\}$. In most of the examples we consider, it suffices to take $c = 1$, which allows us to view each δ_i as an assertion, and generate it automatically using *project&generalize*.

If, when applying rule DISTRANK, the auxiliary constructs h_i and δ_i have no quantifiers, all the resulting premises are $\forall\exists$-premises and the small-model theorem can be used. One of the constructs required to be quantifier free are the helpful assertions, which characterize the set of states from which a given transition is helpful. Many simple protocols have helpful assertions that are quantifier-free (or, with the addition of some auxiliary variables, can be transformed into protocols that have quantifier-free helpful assertions). Some protocols, however, cannot be proven with such restricted assertions. To deal with such protocols, we extend the method of invisible ranking to handle expressions such as $i \pm 1$ to appear both in the transition relation as well as the auxiliary constructs, and helpful assertions (and ranking functions) belonging to transitions of process i to be of the form $h(i) = \forall j.H(i,j)$, where $H(i,j)$ is a quantifier-free assertion.

1.3 Structure of the Thesis

The thesis is organized in the form of an edited collection of the extended versions of six published articles. The first four chapters are associated with systems with pushdown store and the last chapter is associated with parameterized systems.

Chapter 2 is based on [KV00a] and [KPV02a]. As described there, [KPV02a] extends the automata-theoretic approach to infinite-state model checking introduced in [KV00a], and the two papers are now combined in [KPV04]. This chapter discusses the extension of the automata-theoretic framework for infinite-state systems to linear-time specifications. It also includes the proof to the fact that pushdown systems with regular labeling have the same power as prefix-recognizable systems.

Chapter 3 is an extended version of the paper [PV04]. It includes the extension of the automata-theoretic framework for infinite-state systems to handle global model checking. We handle global model checking for both branching-time and linear-time specifications.

Chapter 4 is an extended version of the paper [KPV02b]. We show that model-checking pushdown specifications is decidable over finite-state systems and undecidable over pushdown systems.

Chapter 5 is an extended version of the paper [PV03]. We present the class of micro-macro stack graphs, and give an automata-theoretic solution to the model-checking problem for both branching-time and linear-time specifications.

Chapter 6 is an extended version of the papers [FPPZ04a, FPPZ04b]. This version was submitted to the journal Software Tools for Technology Transfer. We introduce the heuristic to prove liveness properties of parameterized systems using invisible constructs.

Bibliography

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *N. Halbwachs and D. Peled, editors,* Proc. 11^{th} Intl. Conference on Computer Aided Verification (CAV'99), *volume 1633 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 134–145, 1999.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. 10th International Conference on Tools and Algorithms For the Construction and Analysis of Systems*, volume 2725 of *Lecture Notes in Computer Science*, pages 67–79, Barcelona, Spain, April 2004. Springer-Verlag.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. on Foundations of Computer Science*, pages 100–109, Florida, October 1997.

[AJMd02] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *E. Brinksma and K. G.Larsen, editors,* Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02), *volume 2404 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 555–568, 2002.

[AK86] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. 36th ACM Symposium on Theory of Computing*. ACM, ACM press, 2004.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *G. Berry, H. Comon, and A. Finkel, editors,* Proc. 13^{th} Intl. Conference on Computer Aided Verification (CAV'01), *volume 2102 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 221–234, 2001.

[ATM03] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive game graphs. In *Computer-Aided Verification, Proc. 15th International Conference*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2003.

[BBDEL96] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Proc. 33rd Conference on Design Automation*, pages 655–660, Las Vegas, Nevada, USA, June 1996. ACM press.

[BCC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Design Automation Conference*, pages 317–320. IEEE Computer Society, 1999.

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *Proc. 5th ACM Symp. Princ. of Dist. Comp.*, pages 240–248, 1986.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BCMS00] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. Unpublished manuscript, 2000.

[BE96] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1996.

[BEH95] A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *Proc. 10th annual IEEE Symposium on Logic in Computer Science*, pages 123–133, San Diego, CA, USA, June 1995. IEEE computer society press.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, July 1997. Springer-Verlag.

[BER94] A. Bouajjani, R. Echahed, and R. Robbana. Verification of nonregular temporal properties for context-free processes. In *Proc. 5th International Conference on Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 81–97, Uppsala, Sweden, 1994. Springer-Verlag.

[BLM01] P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessors using satisfiability solvers. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.

[BQ96] O. Burkart and Y.-M. Quemener. Model checking of infinite graphs defined by graph grammars. In *Proc. 1st International workshop on verification of infinite states systems*, volume 6 of *ENTCS*, page 15. Elsevier, 1996.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130, Stanford, CA, USA, August 2000. Springer-Verlag.

[BR01] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. 3rd Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 1992.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic J. Comut.*, 2:89–125, 1995.

[BS99] O. Burkart and B. Steffen. Model checking the full modal μ-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[BT02] A. Bouajjani and T. Tuili. Extrapolating tree transfomations. In *E. Brinksma and K. G.Larsen, editors,* Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 539–554, 2002.

[Büc60] J.R. Büchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik und Grundl. Math.*, 6:66–92, 1960.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Bur97a] O. Burkart. Automatic verification of sequential infinite-state processes. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Lecture Notes in Computer Science*, volume 1354. Springer-Verlag, 1997.

[Bur97b] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. In F. Moller, editor, *Proc. 2nd International workshop on verification of infinite states systems*, 1997.

[Cac02] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. In *4th International Workshop on Verification of Infinite-State Systems*, *Electronic Notes in Theoretical Computer Science* 68(6), Brno, Czech Republic, August 2002.

[Cac03] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *Proc. 30th International Colloqium on Automata, Languages, and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569, Eindhoven, The Netherlands, June 2003. Springer-Verlag.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 1996.

[Cau02] D. Caucal. On infinite terms having a decidable monadic theory. In *27th International Symposium on Mathematical Foundations of Computer Science 2002*, volume 2420 of *Lecture Notes in Computer Science*, pages 165–176, Warsaw, Poland, August 2002. Springer-Verlag.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2), 8 1996. Preliminary version appeared in 5th CAV, 1993.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer-Verlag, 2001.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR92)*, volume 962 of *Lect. Notes in Comp. Sci.*, pages 395–407, Philadelphia, PA, August 1995. Springer-Verlag.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CKKV01] H. Chockler, O. Kupferman, R.P. Kurshan, and M.Y. Vardi. A practical approach to coverage in model checking. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 66–78. Springer-Verlag, 2001.

[CKV01] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. In *7th International Conference on Tools and algorithms for the construction and analysis of systems*, number 2031 in Lecture Notes in Computer Science, pages 528 – 542. Springer-Verlag, 2001.

[CR99a] S.J. Creese and A.W. Roscoe. Formal verification of arbitrary network topologies. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, 1999. CSREA Press.

[CR99b] S.J. Creese and A.W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and fdr. In *Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'99)*, Beijing, 1999. Kluwer Academic Publishers.

[CR00] S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, Las Vegas, June 2000. CSREA Press.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[CW98] B. Courcelle and I. Walukiewicz. Monadic second-order logic, graph coverings and unfoldings of transition systems. *Annals of Pure and Applied Logic*, 92(1):35–62, 1998.

[CW02] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proc. 9th ACM conference on Computer and Communications Security*, pages 235–244, Washington, DC, USA, 2002. ACM.

[CW03] A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 2003.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[DW99] M. Dickhfer and T. Wilke. Timed alternating tree automata: the automata-theoretic solution to the TCTL model checking problem. In *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 281–290, Prague, Czech Republic, 1999. Springer-Verlag, Berlin.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247, Chicago, IL, July 2000. Springer-Verlag.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 328–337, White Plains, October 1988.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, San Juan, October 1991.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.

[EKS01] J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 316–339, Sendai, Japan, October 2001. Springer-Verlag.

[Eme87] E.A. Emerson. Uniform inevitability is tree automaton ineffable. *Information Processing Letters*, 24(2):77–79, January 1987.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22nd ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In *R. Alur and T. Henzinger, editors,* Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, 1996.

[ES96] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2), 8 1996. Preliminary version appeared in 5th CAV, 1993.

[ES97] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model checking under fairness assumptions. *ACM Trans. Prog. Lang. Sys.*, 19(4), 1997. Preliminary version appeared in 7th CAV, 1995.

[ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336, Paris, France, July 2001. Springer-Verlag.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In *Proc. 21st Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lect. Notes in Comp. Sci.*, pages 156–170, Bangalore, India, December 2001. Springer-Verlag.

[FPPZ04a] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with incomprehensible ranking. In *Proc. 10^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), volume 2988 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 482–496, April 2004.

[FPPZ04b] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. *Software Tools for Technology Transfer*, 2004. Submitted.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown automata. In F. Moller, editor, *Proc. 2nd International Workshop on Verification of Infinite States Systems*, 1997.

[GGH67a] S. Ginsburg, S.A. Greibach, and M.A. Harrison. One-way stack automata. *Journal of the ACM*, 14(2):381–418, 1967.

[GGH67b] S. Ginsburg, S.A. Greibach, and M.A. Harrison. Stack automata and compiling. *Journal of the ACM*, 14(1):172–201, 1967.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[GS97] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *O. Grumberg, editor, Proc.* Proc. 9^{th} Intl. Conference on Computer Aided Verification, (CAV'97), *volume 1254 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 1997.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, May 1994.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In *B. Steffen, editor, Proc. 4^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 424–438, 1998.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996.

[HKV96] T.A. Henzinger, O. Kupferman, and M.Y. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In *Proc. 7th Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 514–529, Pisa, August 1996. Springer-Verlag.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[ID96] C.N. Ip and D. Dill. Verifying systems with replicated components in Murφ. In *R. Alur and T. Henzinger, editors,* Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96), *volume 1102 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 1996.

[JL98] E. Jensen and N.A. Lynch. A proof of burn's n-process mutual exclusion algorithm using abstraction. In *B. Steffen, editor, Proc. 4^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 409–423, 1998.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *S. Graf and M. Schwartzbach, editors, Proc. 6^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), volume 1785 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 2000.

[KM95] James R. Knight and Eugene W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *O. Grumberg, editor, Proc.* Proc. 9^{th} Intl. Conference on Computer Aided Verification, (CAV'97), *volume 1254 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 424–435, 1997.

[KNU03] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, Grenoble, France, April 2003. Springer-Verlag.

[KP00] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(1):328–342, 2000.

[KPP03] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace containment. In *Computer Aided Verification, Proc. 15th International Conference*, volume 2725 of *Lecture Notes in Computer Science*, pages 381–393. Springer-Verlag, 2003.

[KPV02a] O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 2002.

[KPV02b] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *Proc. 9th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 2514 of *Lecture Notes in Computer Science*, pages 262–277. Springer-Verlag, 2002.

[KPV04] O. Kupferman, N. Piterman, and M.Y. Vardi. Fair equivalence relations. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 702–732. Springer-Verlag, 2004.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[KV98] O. Kupferman and M.Y. Vardi. Modular model checking. In *Proc. Compositionality Workshop*, volume 1536 of *Lecture Notes in Computer Science*, pages 381–401. Springer-Verlag, 1998.

[KV99] O. Kupferman and M.Y. Vardi. Robust satisfaction. In *Proc. 10th Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 383–398. Springer-Verlag, August 1999.

[KV00a] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2000.

[KV00b] O. Kupferman and M.Y. Vardi. Synthesis with incomplete informatio. In *Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, January 2000.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LS97] D. Lesens and H. Saidi. Automatic verification of parameterized networks of processes by abstraction. In *2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, 1997.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mor00] C. Morvan. On rational graphs. In *Proc. 3rd International Conference on Foundations of Software Science and Computation Structures*, volume 1784 of *Lecture Notes in Computer Science*, pages 252–266, Berlin, Germany, March 2000. Springer-Verlag.

[MP91] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP95] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety.* Springer-Verlag, New York, 1995.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[PI95] W. Peng and S. P. Iyer. A new typee of pushdown automata on infinite tree. *International Journal of Foundations of Computer Science*, 6(2):169–186, June 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, Austin, January 1989.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97, Genova, Italy, April 2001. Springer-Verlag.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *A. Emerson and P. S. Sistla, editors,* Proc. 12^{th} Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 328–343, 2000.

[PV03] N. Piterman and M.Y. Vardi. From bidirectionality to alternation. *Theoretical Computer Science*, 295(1–3):295–321, February 2003.

[PV04] N. Piterman and M. Vardi. Global model-checking of infinite-state systems. In *Proc. 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–400. Springer-Verlag, 2004.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[SCFG84] A. Sistla, E.M. Clarke, N. Francez, and Y. Gurevich. Can message buffers be axiomatized in linear temporal logic. *Information and Control*, 63(1/2):88–112, 1984.

[Sch02] S. Schwoon. *Model-checking pushdown systems.* PhD thesis, Technische Universität München, 2002.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 151–165. Springer-Verlag, 1989.

[Sha00] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *Proc. 11th International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16, University Park, PA, USA, August 2000. Springer-Verlag.

[SOR93] R.E. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (beta release). Technical report, Computer Science laboratory, SRI International, Menlo Park, California, March 1993.

[SW91] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[Tho01] W. Thomas. A short introduction to infinite automata. In *Proc. 5th. international conference on Developments in Language Theory*, volume 2295 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, July 2001.

[VB00] W. Visser and H. Barringer. Practical CTL* model checking: Should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer-Verlag, 1996.

[Wal02] I. Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275(1-2):311–346, March 2002.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems, Proc. International Workshop, Grenoble*, volume 407, pages 68–80, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 185–194, Tucson, 1983.

[WW96] B. Willems and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Proc. 11th Symp. on Logic in Computer Science*, pages 294–303, New Brunswick, July 1996.

Chapter 2

An Automata-Theoretic Approach to Infinite-State Systems

We develop an automata-theoretic framework for reasoning about infinite-state sequential systems. Our framework is based on the observation that states of such systems, which carry a finite but unbounded amount of information, can be viewed as nodes in an infinite tree, and transitions between states can be simulated by finite-state automata. Checking that the system satisfies a temporal property can then be done by an alternating two-way tree automaton that navigates through the tree. We show how this framework can be used to solve the model-checking problem for μ-calculus and LTL specifications with respect to pushdown and prefix-recognizable systems.

We show that we cannot reduce LTL model-checking to μ-calculus model-checking. In order to handle model-checking of linear-time specifications we introduce *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree. We study the expressive power of path automata and some of the decision problems related to them.

Our framework also handles systems with regular labeling, and in fact we show that model-checking with respect to pushdown systems with regular labeling is intereducible with model-checking with respect to prefix-recognizable systems with simple labeling.

As has been the case with finite-state systems, the automata-theoretic framework is quite versatile. We demonstrate it by solving the realizability and synthesis problems for μ-calculus specifications with respect to prefix-recognizable environments, and explaining how to extend our framework to handle systems with *regular fairness constraints* and μ-calculus with *backward modalities*.

2.1 Introduction

One of the most significant developments in the area of formal design verification is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [CES86, LP85, QS81, VW86a]. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior (for a survey, see [CGP99]). Symbolic methods that enable model checking of very large state spaces, and the great ease of use of fully algorithmic methods, led to industrial acceptance of temporal model checking [BLM01, CFF$^+$01].

An important research topic over the past decade has been the application of model checking to infinite-state systems. Notable success in this area has been the application of model checking to real-time and hybrid systems (cf. [HHWT95, LPY97]). Another active thrust of research is the application of model checking to *infinite-state sequential systems*. These are systems in which a state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The origin of this thrust is the important result by Müller and Schupp that the monadic second-order theory of *context-free graphs* is decidable [MS85]. As the complexity involved in that decidability result is nonelementary, researchers sought decidability results of elementary complexity. This started with Burkart and Steffen, who developed an exponential-time algorithm for model-checking formulas in the *alternation-free* μ-calculus with respect to context-free graphs [BS92]. Researchers then went on to extend this result to the μ-calculus, on one hand, and to more general graphs on the other hand, such as *pushdown graphs* [BS95, Wal96], *regular graphs* [BQ96], and *prefix-recognizable graphs* [Cau96]. The most powerful results so far are algorithms for model-checking formulas of μ-calculus with respect to prefix-recognizable graphs in exponential-time [Bur97b] and with respect to *high order pushdown graphs* in nonelementary time [KNU03, Cac03]. See also [BS95, Cau96, BE96, BEM97, Bur97a, FWW97, BS99, BCMS00, CW03]. Some of this theory has also been reduced to practice. Pushdown model-checkers such as Mops [CW02], Moped [ES01, Sch02], and Bebop [BR00] (to name a few) have been developed. Of the mentioned three, the industrial application, Bebop, enables only model checking of safety properties. Successful applications of these model-checkers to the verification of software are reported, for example, in [BR01, CW02].

In this paper, we develop an automata-theoretic framework for reasoning about infinite-state sequential systems. The automata-theoretic approach uses the theory of automata as a unifying paradigm for system specification, verification, and synthesis [WVS83, EJ91, Kur94, VW94, KVW00]. Automata enable the separation of the logical and the algorithmic aspects of reasoning about systems, yielding clean and asymptotically optimal algorithms. Automata are the key to techniques such as on-the-fly verification [GPVW95], and they are useful also for modular verification [KV98], partial-order verification [GW94, WW96], verification of real-time and hybrid systems [HKV96, DW99], and verification of open systems [AHK97, KV99]. Many decision and synthesis problems have automata-based solutions and no other solution for them is known [EJ88, PR89, KV00b]. Automata-based methods have been implemented in industrial automated-verification tools (c.f., COSPAN [HHK96] and SPIN [Hol97, VB00]).

The automata-theoretic approach, however, has long been thought to be inapplicable for effective reasoning about infinite-state systems. The reason, essentially, lies in the fact that the automata-theoretic techniques involve constructions in which the state space of the system directly influences the state space of the automaton (e.g., when we take the product of a specification automaton with the graph that models the system). On the other hand, the automata we know to handle have finitely many states. The key insight, which enables us to overcome this difficulty, and which is implicit in all previous decidability results in the area of infinite-state sequential systems, is that in spite of the somewhat misleading terminology (e.g., "context-free graphs" and "pushdown graphs"), the classes of infinite-state graphs for which decidability is known can be described by finite-state automata. This is explained by the fact the the states of the graphs that model these systems can be viewed as nodes in an infinite tree and transitions between states can be expressed by finite-state automata. As a result, automata-theoretic techniques can be used to reason about such systems. In particular, we show that various problems related to the analysis of such systems can be reduced to the membership and emptiness problems for *alternating two-way tree automata*, which was shown to be decidable in exponential time [Var98].

We first show how the automata-theoretic framework can be used to solve the μ-calculus model-checking problem with respect to pushdown and prefix-recognizable systems. As explained, the solution is based on the observation that states of such systems correspond to a location in an infinite tree. Transitions of the system, can be simulated by a finite state automaton that reads the infinite tree. Thus, the model-checking problem of μ-calculus over pushdown and prefix-recognizable graphs is reduced to the membership problem of 2-way alternating parity tree automata, namely, the question whether an automaton accepts the tree obtained by unwinding a given finite labeled graph. The complexity of our algorithm matches the complexity of previous algorithms.

The μ-calculus is sufficiently strong to express all properties expressible in the linear temporal logic LTL (and in fact, all properties expressible by an ω-regular language) [Dam94]. Thus, by translating LTL formulas into μ-calculus formulas we can use our solution for μ-calculus model checking in order to solve LTL model checking. This solution, however, is not optimal. This has to do both with the fact that the translation of LTL to μ-calculus is exponential, as well as the fact that our solution for μ-calculus model checking is based on tree automata. A tree automaton splits into several copies when it runs on a tree. While splitting is essential for reasoning about branching properties, it has a computational price. For linear properties, it is sufficient to follow a single computation of the system, and tree automata seem too strong for this task. For example, while the application of the framework developed above to pushdown systems and LTL properties results in an algorithm that is doubly-exponential in the formula and exponential in the system, the problem is known to be EXPTIME-complete in the formula and polynomial in the system [BEM97].

In order to handle model checking of linear-time properties, we introduce *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree. In particular, *two-way* nondeterministic path automata enable exactly the type of navigation that is required in order to check linear properties of infinite-state sequential systems. We study the expressive power and the complexity of the decision problems for (two way) path automata. The fact that path automata follow a single path in the tree makes them very similar to two-way nondeterministic automata on infinite words. This enables us to reduce the membership problem (whether an automaton accepts the tree obtained by unwinding a given finite labeled graph) of two-way nondeterministic path automata to the emptiness problem of one-way alternating Büchi automata on infinite words, which was studied in [VW86b]. This leads to a quadratic upper bound for the membership problem for two-way nondeterministic path automata.

Using path automata we are able to solve the problem of LTL model checking with respect to pushdown and prefix-recognizable systems by a reduction to the membership problem of two-way nondeterministic path automata. Usually, automata-theoretic solutions to model checking use the emptiness problem, namely whether an automaton accepts some tree. We note that for (linear-time) model checking of sequential infinite-state system both simplifications, to the membership problem vs. the emptiness problem, and to path automata vs. tree automata are crucial: as we prove the emptiness problem for two-way nondeterministic Büchi path automata is EXPTIME-complete, and the membership problem for two-way alternating Büchi tree automata is also EXPTIME-complete[1]. Our automata-theoretic technique matches the known upper bound for model-checking LTL properties on pushdown systems [BEM97, EHRS00]. In addition, the automata-theoretic

[1] In contract, the membership problem for one-way alternating Büchi tree automata can be reduced to the emptiness problem of the 1-letter alternating word automaton obtained by taking the product of the labeled graph that models the tree with the one-way alternating tree automaton [KVW00]. This technique cannot be applied to two-way

approach provides the first solution for the case where the system is prefix-recognizable. Specifically, we show that we can solve the model-checking problem of an LTL formula φ with respect to a prefix-recognizable system R of size n in time and space $2^{O(n+|\varphi|)}$. We also prove a matching EXPTIME lower bound.

Usually, the labeling of the state depends on the internal state of the system and the top of the store. Our framework also handles *regular labeling*, where the label depends on whether the word on the store is a member in some regular language. The complexity is exponential in the nondeterministic automata that describe the labeling, matching the known bound for pushdown systems and linear-time specifications [EKS01]. The automata-theoretic techniques for handling regular labeling and for handling the regular transitions of a prefix-recognizable system are very similar. This leads us to the understanding that regular labeling and prefix-recognizability have exactly the same power. Formally, we prove that model checking (for either μ-calculus or LTL) with respect to a prefix-recognizable system can be reduced to model checking with respect to a pushdown system with regular labeling, and vice versa. For linear-time properties, it is known that model checking of a pushdown system with regular labeling is EXPTIME-complete [EKS01]. Hence, our reductions suggest an alternative proof of the exponential upper and lower bounds for the problem of LTL model checking of prefix-recognizable systems.

While most of the complexity results established for model checking of infinite-state sequential systems using our framework are not new, it appears to be, like the automata-theoretic framework for finite-state systems, very versatile, and it has further potential applications. We proceed by showing how to solve the *realizability* and *synthesis* problem of μ-calculus formulas with respect to infinite-state sequential environments. This was later extended to LTL formulas in [ATM03]. We then discuss how to extend the algorithms to handle graphs with *regular fairness constraints*, and to μ-calculus with *backward modalities*. In both these problems all we demonstrate is a (fairly simple) extension of the basic algorithm; the (exponentially) hard work is then done by the membership-checking algorithm. The automata-theoretic framework for reasoning about infinite-state sequential systems was also extended to global model checking [PV04] and to classes of systems that are more expressive than prefix-recognizable [Cac03, PV03]. It can be easily extended to handle also CARET specifications [AEM04].

Extended abstracts of the work presented in this paper appeared in [KV00a, KPV02].

2.2 Preliminaries

Given a finite set Σ, a *word* over Σ is a finite or infinite sequence of symbols from Σ. We denote by Σ^* the set of finite sequences over Σ and by Σ^ω the set of infinite sequences over Σ. Given a word $w = \sigma_0\sigma_1\sigma_2\cdots \in \Sigma^* \cup \Sigma^\omega$, we denote by $w_{\geq i}$ the suffix of w starting at σ_i, i.e., $w_{\geq i} = \sigma_i\sigma_{i+1}\cdots$. The *length* of w is denoted by $|w|$ and is defined to be ω for infinite words.

2.2.1 Labeled Transition Graphs and Rewrite Systems

A *labeled transition graph* is $G = \langle \Sigma, S, L, \rho, s_0 \rangle$, where Σ is a finite set of labels, S is a (possibly infinite) set of states, $L : S \to \Sigma$ is a labeling function, $\rho \subseteq S \times S$ is a transition relation, and $s_0 \in S_0$ is an initial state. When $\rho(s, s')$, we say that s' is a *successor* of s, and s is a *predecessor* of

automata, since they can distinguish between a graph and its unwinding. For a related discussion regarding past-time connectives in branching temporal logics, see [KP95].

s'. For a state $s \in S$, we denote by $G^s = \langle \Sigma, S, L, \rho, s \rangle$, the graph G with s as its initial state. An *s-computation* is an infinite sequence of states $s_0, s_1, \ldots \in S^\omega$ such that $s_0 = s$ and for all $i \geq 0$, we have $\rho(s_i, s_{i+1})$. An s-computation $s_0, s_1, \ldots$ induces the *s-trace* $L(s_0) \cdot L(s_1) \cdots$. Let $\mathcal{T}_s$ be the set of all s-traces.

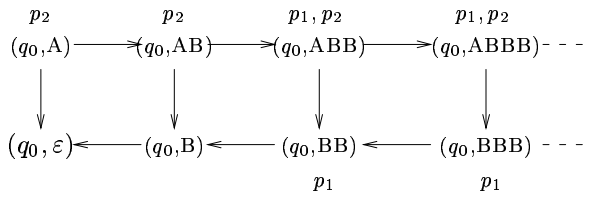
A *rewrite system* is $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$, where Σ is a finite set of labels, V is a finite alphabet, Q is a finite set of states, $L : Q \times V^* \to \Sigma$ is a labeling function, T is a finite set of rewrite rules, to be defined below, q_0 is an initial state, and $x_0 \in V^*$ is an initial word. The set of *configurations* of the system is $Q \times V^*$. Intuitively, the system has finitely many control states and an unbounded store. Thus, in a configuration $(q, x) \in Q \times V^*$ we refer to q as the *control state* and to x as the *store*. A configuration $(q, x) \in Q \times V^*$ indicates that the system is in control state q with store x. We consider here two types of rewrite systems. In a *pushdown* system, each rewrite rule is $\langle q, A, x, q' \rangle \in Q \times V \times V^* \times Q$. Thus, $T \subseteq Q \times V \times V^* \times Q$. In a *prefix-recognizable* system, each rewrite rule is $\langle q, \alpha, \beta, \gamma, q' \rangle \in Q \times reg(V) \times reg(V) \times reg(V) \times Q$, where $reg(V)$ is the set of regular expressions over V. Thus, $T \subseteq Q \times reg(V) \times reg(V) \times reg(V) \times Q$. For a word $w \in V^*$ and a regular expression $r \in reg(V)$ we write $w \in r$ to denote that w is in the language of the regular expression r. We note that the standard definition of prefix-recognizable systems does not include control states. Indeed, a prefix-recognizable system without states can simulate a prefix-recognizable system with states by having the state as the first letter of the unbounded store. We use prefix-recognizable systems with control states for the sake of uniform notation.

We consider two types of labeling functions, *simple* and *regular*. The labeling function associates with a configuration $(q, x) \in Q \times V^*$ a symbol from Σ. A simple labeling function depends only on the first letter of x. Thus, we may write $L : Q \times (V \cup \{\epsilon\}) \to \Sigma$. Note that the label is defined also for the case that x is the empty word ϵ. A regular labeling function considers the entire word x but can only refer to its membership in some regular set. Formally, for every state q there is a partition of V^* to $|\Sigma|$ regular languages $R_1, \ldots R_{|\Sigma|}$, and $L(q, x)$ depends on the regular set that x belongs to. For a letter $\sigma \in \Sigma$ and a state $q \in Q$ we set $R_{\sigma,q} = \{x \mid L(q, x) = \sigma\}$ to be the regular language of store contents that produce the label σ (with state q). We are especially interested in the cases where the alphabet Σ is the powerset 2^{AP} of the set of atomic propositions. In this case, we associate with every state q and proposition p a regular language $R_{p,q}$ that contains all the words w for which the proposition p is true in configuration (q, x). Thus $p \in L(q, x)$ iff $x \in R_{p,q}$. Unless mentioned explicitly, the system has a simple labeling.

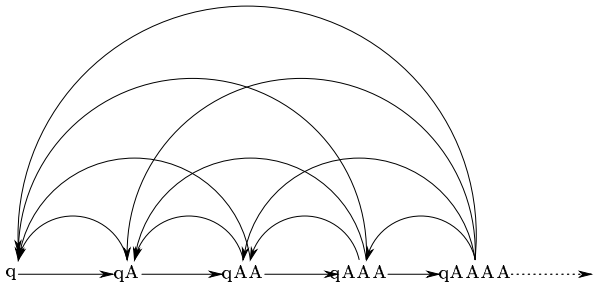
The rewrite system R induces the labeled transition graph $G_R = \langle \Sigma, Q \times V^*, L, \rho_R, (q_0, x_0) \rangle$. The states of G_R are the configurations of R and $\langle (q, z), (q', z') \rangle \in \rho_R$ if there is a rewrite rule $t \in T$ leading from configuration (q, z) to configuration (q', z'). Formally, if R is a pushdown system, then $\rho_R((q, A \cdot y), (q', x \cdot y))$ if $\langle q, A, x, q' \rangle \in T$; and if R is a prefix-recognizable system, then $\rho_R((q, x \cdot y), (q', x' \cdot y))$ if there are regular expressions α, β, and γ such that $x \in \alpha$, $y \in \beta$, $x' \in \gamma$, and $\langle q, \alpha, \beta, \gamma, q' \rangle \in T$. Note that in order to apply a rewrite rule in state $(q, z) \in Q \times V^*$ of a pushdown graph, we only need to match the state q and the first letter of z with the second element of a rule. On the other hand, in an application of a rewrite rule in a prefix-recognizable graph, we have to match the state q and we should find a partition of z to a prefix that belongs to the second element of the rule and a suffix that belongs to the third element. A labeled transition graph that is induced by a pushdown system is called a *pushdown graph*. A labeled transition system that is induced by a prefix-recognizable system is called a *prefix-recognizable graph*.

Example 2.2.1 *Consider the pushdown system* $\langle 2^{\{p_1,p_2\}}, \{A, B\}, \{q_0\}, L, T, q_0, A \rangle$, *with transition* $T = \{\langle q_0, A, AB, q_0 \rangle, \langle q_0, A, \varepsilon, q_0 \rangle, \langle q_0, B, \varepsilon, q_0 \rangle\}$, *and* L *defined by* $R_{q_0,p_1} = \{A, B\}^* \cdot B \cdot B \cdot \{A, B\}^*$

and $R_{q_0,p_2} = A \cdot \{A, B\}^*$*, induces the labeled transition graph below.*



Example 2.2.2 *Consider the prefix-recognizable system* $\langle 2^{\emptyset}, \{A\}, \{q\}, L, T, q_0, A\rangle$*, with transition* $T = \{\langle q, A^*, A^*, \varepsilon, q\rangle, \langle q, \varepsilon, A^*, A, q\rangle\}$ *induces the labeled transition graph below.*



Consider a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0\rangle$. For a rewrite rule $t_i = \langle s,\ \alpha_i,\ \beta_i,\ \gamma_i,\ s'\rangle \in T$, let $\mathcal{U}_\lambda = \langle V, Q_\lambda, \eta_\lambda, q^0_\lambda, F_\lambda\rangle$, for $\lambda \in \{\alpha_i, \beta_i, \gamma_i\}$, be the nondeterministic automaton for the language of the regular expression λ. We assume that all initial states have no incoming edges and that all accepting states have no outgoing edges. We collect all the states of all the automata for α, β, and γ regular expressions. Formally, $Q_\alpha = \bigcup_{t_i \in T} Q_{\alpha_i}$, $Q_\beta = \bigcup_{t_i \in T} Q_{\beta_i}$, and $Q_\gamma = \bigcup_{t_i \in T} Q_{\gamma_i}$. We assume that we have an automaton whose language is $\{x_0\}$. We denote the final state of this automaton by x_0 and add all its states to Q_γ. Finally, for a regular labeling function L, a state $q \in Q$, and a letter $\sigma \in \Sigma$, let $\mathcal{U}_{\sigma,q} = \langle V, Q_{\sigma,q}, q^0_{\sigma,q}, \rho_{\sigma,q}, F_{\sigma,q}\rangle$ be the nondeterministic automaton for the language $R_{\sigma,q}$. In a similar way given a state $q \in Q$ and a proposition $p \in AP$, let $\mathcal{U}_{p,q} = \langle V, Q_{p,q}, q^0_{p,q}, \rho_{p,q}, F_{p,q}\rangle$ be the nondeterministic automaton for the language $R_{p,q}$.

We define the *size* $\|T\|$ of T as the space required in order to encode the rewrite rules in T. Thus, in a pushdown system, $\|T\| = \sum_{\langle q,A,x,q'\rangle \in T} |x|$, and in a prefix-recognizable system, $\|T\| = \sum_{\langle q,\alpha,\beta,\gamma,q'\rangle \in T} |\mathcal{U}_\alpha| + |\mathcal{U}_\beta| + |\mathcal{U}_\gamma|$. In the case of a regular labeling function, we also measure the labeling function $\|L\| = \sum_{q \in Q} \sum_{\sigma \in \Sigma} |\mathcal{U}_{\sigma,q}|$ or $\|L\| = \sum_{q \in Q} \sum_{p \in AP} |\mathcal{U}_{p,q}|$.

2.2.2 μ-Calculus

We give a short introduction to μ-calculus [Koz83]. The *μ-calculus* is a modal logic augmented with least and greatest fixpoint operators. Given a finite set AP of atomic propositions and a finite set *Var* of variables, a μ-calculus formula (in a positive normal form) over AP and *Var* is one of the following:

- **true**, **false**, p for all $p \in AP$, or y for all $y \in Var$;
- $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, for μ-calculus formulas φ_1 and φ_2;

- $\Box\varphi$ or $\Diamond\varphi$ for a μ-calculus formula φ.
- $\mu y.\varphi$ or $\nu y.\varphi$, for $y \in \mathit{Var}$ and a μ-calculus formula φ.

A *sentence* is a formula that contains no free variables from *Var* (that is, every variable is in the scope of some fixed-point operator that binds it). We define the semantics of μ-calculus with respect to a labeled transition graph $G = \langle 2^{AP}, S, L, \rho, s_0\rangle$ and a valuation $\mathcal{V} : \mathit{Var} \to 2^S$. Each formula ψ and valuation $\mathcal{V}$ then define a set $[[\psi]]^G_{\mathcal{V}}$ of states of G that satisfy the formula. For a valuation $\mathcal{V}$, a variable $y \in \mathit{Var}$, and a set $S' \subseteq S$, we denote by $\mathcal{V}[y \leftarrow S']$ the valuation obtained from $\mathcal{V}$ by assigning S' to y. The mapping $[[\psi]]^G_{\mathcal{V}}$ is defined inductively as follows:

- $[[\mathbf{true}]]^G_{\mathcal{V}} = S$ and $[[\mathbf{false}]]^G_{\mathcal{V}} = \emptyset$;
- For $y \in \mathit{Var}$, we have $[[y]]^G_{\mathcal{V}} = \mathcal{V}(y)$;
- For $p \in AP$, we have $[[p]]^G_{\mathcal{V}} = \{s \mid p \in L(s)\}$;
- $[[\psi_1 \wedge \psi_2]]^G_{\mathcal{V}} = [[\psi_1]]^G_{\mathcal{V}} \cap [[\psi_2]]^G_{\mathcal{V}}$;
- $[[\psi_1 \vee \psi_2]]^G_{\mathcal{V}} = [[\psi_1]]^G_{\mathcal{V}} \cup [[\psi_2]]^G_{\mathcal{V}}$;
- $[[\Box\psi]]^G_{\mathcal{V}} = \{s \in S :$ for all s' such that $\rho(s, s'),$ we have $s' \in [[\psi]]^G_{\mathcal{V}}\}$;
- $[[\Diamond\psi]]^G_{\mathcal{V}} = \{s \in S :$ there is s' such that $\rho(s, s')$ and $s' \in [[\psi]]^G_{\mathcal{V}}\}$;
- $[[\mu y.\psi]]^G_{\mathcal{V}} = \bigcap\{S' \subseteq S : [[\psi]]^G_{\mathcal{V}[y\leftarrow S']} \subseteq S'\}$;
- $[[\nu y.\psi]]^G_{\mathcal{V}} = \bigcup\{S' \subseteq S : S' \subseteq [[\psi]]^G_{\mathcal{V}[y\leftarrow S']}\}$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. For a full exposition of μ-calculus we refer the reader to [Eme97].

Note that $[[\psi]]^G_{\mathcal{V}}$ depends only on the valuation of free variables in ψ. In particular, no valuation is required for a sentence and we write $[[\psi]]^G$. For a state $s \in S$ and a sentence ψ, we say that ψ holds at s in G, denoted $G, s \models \psi$ iff $s \in [[\psi]]^G$. Also, $G \models \psi$ iff $G, s_0 \models \psi$. We say that a rewrite system R satisfies a μ-calculus formula ψ if $G_R \models \psi$.

The *model-checking problem* for a labeled transition graph G and a μ-calculus formula ψ is to determine whether G satisfies ψ.

Theorem 2.2.3 *The model-checking problem for a rewrite system R and a μ-calculus formula φ is solvable in time $2^{O(\|T\|\cdot|\psi|\cdot k)}$ where k is the alternation depth of ψ* [Wal96, Bur97b].

2.2.3 Linear Temporal Logic

We give a short introduction to linear temporal logic (*LTL*) [Pnu77]. LTL augments Boolean arithmetic with temporal quantifiers. Given a finite set AP of propositions, an LTL formula is one of the following.

- **true**, **false**, p for all $p \in AP$;
- $\neg\varphi_1,\ \varphi_1 \vee \varphi_2,\ \varphi_1 \wedge \varphi_2,\ \bigcirc\varphi_1$ and $\varphi_1 U \varphi_2$, for LTL formulas φ_1 and φ_2;

The semantics of LTL formulas is defined with respect to an infinite sequence $\pi \in (2^{AP})^\omega$ and a location $i \in \mathbb{N}$. We use $(\pi, i) \models \psi$ to indicate that the word π in the designated location i satisfies the formula ψ.

- For a proposition $p \in AP$, we have $(\pi, i) \models p$ iff $p \in \pi_i$;
- $(\pi, i) \models \neg f_1$ iff not $(\pi, i) \models f_1$;
- $(\pi, i) \models f_1 \vee f_2$ iff $(\pi, i) \models f_1$ or $(\pi, i) \models f_2$;
- $(\pi, i) \models f_1 \wedge f_2$ iff $(\pi, i) \models f_1$ and $(\pi, i) \models f_2$;
- $(\pi, i) \models \bigcirc f_1$ iff $(\pi, i+1) \models f_1$;
- $(\pi, i) \models f_1 U f_2$ iff there exists $k \geq i$ such that $(\pi, k) \models f_2$ and for all $i \leq j < k$, we have $(\pi, j) \models f_1$;

If $(\pi, 0) \models \psi$ we say that π *satisfies* ψ. We denote by $L(\psi)$ the set of sequences π that satisfy ψ. Given a graph G and a state s of G, we say that s satisfies an LTL formula φ, denoted $(G, s) \models \varphi$, iff $\mathcal{T}_s \subseteq \mathcal{L}(\varphi)$. A graph G satisfies an LTL formula φ, denoted $G \models \varphi$, iff its initial state satisfies it; that is $(G, s_0) \models \varphi$.

The *model-checking problem* for a labeled transition graph G and an LTL formula φ is to determine whether G satisfies φ. Note that the transition relation of R need not be total. There may be finite paths but satisfaction is determined only with respect to infinite paths. In particular, if the graph has only finite paths, its set of traces is empty and the graph satisfies every LTL formula [2]. We say that a rewrite system R satisfies an LTL formula φ if $G_R \models \varphi$. [3]

Theorem 2.2.4 *The model-checking problem for a pushdown system R and an LTL formula φ is solvable*

- *in time $\|T\|^3 \cdot 2^{O(|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|)}$ in the case that L is a simple labeling function* [EHRS00].
- *in time $\|T\|^3 \cdot 2^{O(\|L\|+|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(\|L\|+|\varphi|)}$ in the case that L is a regular labeling function. The problem is EXPTIME-hard in $\|L\|$ even for a fixed formula* [EKS01].

2.2.4 Alternating Two-Way Automata

Given a finite set Υ of directions, an Υ-*tree* is a set $T \subseteq \Upsilon^*$ such that if $v \cdot x \in T$, where $v \in \Upsilon$ and $x \in \Upsilon^*$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $v \in \Upsilon$ and $x \in T$, the node x is the *parent* of $v \cdot x$. Each node $x \neq \varepsilon$ of T has a *direction* in Υ. The direction of the root is the symbol $\perp$ (we assume that $\perp \notin \Upsilon$). The direction

[2] It is also possible to consider finite paths. In this case, the nondeterministic Büchi automaton in Theorem 2.2.8 has to be modified so that it can recognize also finite words (cf. [GO03]). Our results are easily extended to consider also finite paths.

[3] Some work on verification of infinite-state system (e.g., [EHRS00]), consider properties given by nondeterministic Büchi word automata, rather than LTL formulas. Since we anyway translate LTL formulas to automata, we can easily handle also properties given by automata.

of a node $v \cdot x$ is v. We denote by $dir(x)$ the direction of node x. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $v \in \Upsilon$ such that $v \cdot x \in \pi$. Note that our definitions here reverse the standard definitions (e.g., when $\Upsilon = \{0, 1\}$, the successors of the node 0 are 00 and 10 (rather than 00 and 01)[4].

Given two finite sets Υ and Σ, a Σ-*labeled* Υ-*tree* is a pair $\langle T, \tau \rangle$ where T is an Υ-tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, \tau \rangle$ a labeled tree. We say that an $((\Upsilon \cup \{\bot\}) \times \Sigma)$-labeled Υ-tree $\langle T, \tau \rangle$ is Υ-*exhaustive* if for every node $x \in T$, we have $\tau(x) \in \{dir(x)\} \times \Sigma$.

A labeled tree is *regular* if it is the unwinding of some finite labeled graph. More formally, a *transducer* $\mathcal{D}$ is a tuple $\langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, where Υ is a finite set of directions, Σ is a finite alphabet, Q is a finite set of states, $\eta : Q \times \Upsilon \to Q$ is a deterministic transition function, $q_0 \in Q$ is a start state, and $L : Q \to \Sigma$ is a labeling function. We define $\eta : \Upsilon^* \to Q$ in the standard way: $\eta(\varepsilon) = q_0$ and $\eta(ax) = \eta(\eta(x), a)$. Intuitively, a transducer is a labeled finite graph with a designated start node, where the edges are labeled by Υ and the nodes are labeled by Σ. A Σ-labeled Υ-tree $\langle \Upsilon^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{D} = \langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, such that for every $x \in \Upsilon^*$, we have $\tau(x) = L(\eta(x))$. The size of $\langle \Upsilon^*, \tau \rangle$, denoted $\|\tau\|$, is $|Q|$, the number of states of $\mathcal{D}$.

Alternating automata on infinite trees generalize nondeterministic tree automata and were first introduced in [MS87]. Here we describe alternating *two-way* tree automata. For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**, and, as usual, $\wedge$ has precedence over $\vee$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning **true** to elements in Y and assigning **false** to elements in $X \setminus Y$ makes θ true. For a set Υ of directions, the *extension* of Υ is the set $ext(\Upsilon) = \Upsilon \cup \{\varepsilon, \uparrow\}$ (we assume that $\Upsilon \cap \{\varepsilon, \uparrow\} = \emptyset$). An *alternating two-way automaton* over Σ-labeled Υ-trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(ext(\Upsilon) \times Q)$ is the transition function, $q_0 \in Q$ is an initial state, and F specifies the acceptance condition.

A run of an alternating automaton $\mathcal{A}$ over a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a labeled tree $\langle T_r, r \rangle$ in which every node is labeled by an element of $\Upsilon^* \times Q$. A node in T_r, labeled by (x, q), describes a copy of the automaton that is in the state q and reads the node x of Υ^*. Note that many nodes of T_r can correspond to the same node of Υ^*; there is no one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its successors have to satisfy the transition function. Formally, a run $\langle T_r, r \rangle$ is a Σ_r-labeled Γ-tree, for some set Γ of directions, where $\Sigma_r = \Upsilon^* \times Q$ and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\varepsilon, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, \tau(x)) = \theta$. Then there is a (possibly empty) set $S \subseteq ext(\Upsilon) \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and the following hold:
   - If $c \in \Upsilon$, then $r(\gamma \cdot y) = (c \cdot x, q')$.
   - If $c = \varepsilon$, then $r(\gamma \cdot y) = (x, q')$.
   - If $c = \uparrow$, then $x = v \cdot z$, for some $v \in \Upsilon$ and $z \in \Upsilon^*$, and $r(\gamma \cdot y) = (z, q')$.

[4] As will get clearer in the sequel, the reason for that is that rewrite rules refer to the prefix of words.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $c = \uparrow$, we require that $x \neq \varepsilon$.

A run $\langle T_r, r\rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. We consider here *parity* acceptance conditions [EJ91]. A parity condition over a state set Q is a finite sequence $F = \{F_1, F_2, \ldots, F_m\}$ of subsets of Q, where $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_m = Q$. The number m of sets is called the *index* of $\mathcal{A}$. Given a run $\langle T_r, r\rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in \Upsilon^* \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies the condition F if there is an even i for which $inf(\pi) \cap F_i \neq \emptyset$ and $inf(\pi) \cap F_{i-1} = \emptyset$. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. The automaton $\mathcal{A}$ is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$. *Büchi* acceptance condition [Büc62] is a private case of parity of index 3. Büchi condition $F \subseteq Q$ is equivalent to parity condition $\langle \emptyset, F, Q\rangle$. A path π satisfies Büchi condition F iff $inf(\pi) \cap F \neq \emptyset$. *Co-Büchi* acceptance condition is the dual of Büchi. Co-Büchi condition $F \subseteq Q$ is equivalent to parity condition $\langle F, Q\rangle$. A path π satisfies co-Büchi condition F iff $inf(\pi) \cap F = \emptyset$.

The size of an automaton is determined by the number of its states and the size of its transition function. The size of the transition function is $\eta = \Sigma_{q\in Q}\Sigma_{\sigma\in\Sigma}|\eta(q,a)|$ where, for a formula in $B^+(ext(\Upsilon) \times Q)$ we define $|(\Delta, q)| = |\mathbf{true}| = |\mathbf{false}| = 1$ and $|\theta_1 \vee \theta_2| = |\theta_1 \wedge \theta_2| = |\theta_1| + |\theta_2| + 1$.

We say that $\mathcal{A}$ is advancing if δ is restricted to formulas in $B^+((\Upsilon \cup \{\varepsilon\}) \times Q)$, it is one-way if δ is restricted to formulas in $B^+(\Upsilon \times Q)$. We say that $\mathcal{A}$ is nondeterministic if its transitions are of the form $\bigvee_{i\in I}\bigwedge_{v\in\Upsilon}(v, q_v^i))$, in such cases we write $\delta : Q \times \Sigma \to 2^{Q^{|\Upsilon|}}$. In particular, a nondeterministic automaton is 1-way. It is easy to see that a run tree of a nondeterministic tree automaton visits every node in the input tree exactly once. Hence, a run of a nondeterministic tree automaton on tree $\langle T, \tau\rangle$ is $\langle T, r\rangle$ where $r : T \to Q$. Note, that τ and r use the same domain T. In the case that $|\Upsilon| = 1$, $\mathcal{A}$ is a word automaton. In the run of a word automaton, the location of the automaton on the word is identified by the length of its location. Hence, instead of marking the location by v^i, we mark it by i. Formally, a run of a word automaton is $\langle T, r\rangle$ where $r : T \to \mathbb{N} \times Q$ and a node $x \in T$ such that $r(x) = (i, q)$ signifies that the automaton in state q is reading the ith letter of the word. In the case of word automata, there is only one direction $v \in \Upsilon$. Hence, we replace the atoms $(d, q) \in ext(\Upsilon) \times Q$ in the transition of $\mathcal{A}$ by atoms from $\{-1, 0, 1\} \times Q$ where -1 means read the previous letter, 0 means read again the same letter, and 1 means read the next letter. Accordingly, the pair $(i, q), (j, q')$ satisfies the transition of $\mathcal{A}$ if there exists $(d, q') \in \delta(q, w_i)$ such that $j = i + d$. In the case that the automaton is 1-way the length of x uniquely identifies the location in the word. That is, we use $r : T \to Q$ and $r(x) = q$ signifies that state q is reading letter $|x|$. In the case that a word automaton is nondeterministic, its run is an infinite sequence of locations and states. Namely, $r = (0, q_0), (i_1, q_1), \ldots$. In addition, if the automaton is 1-way the location in the sequence identifies the letter read by the automaton and we write $r = q_0, q_1, \ldots$.

Theorem 2.2.5 *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, we can construct an equivalent nondeterministic one-way parity tree automaton whose number of states is exponential in nk and whose index is linear in nk* [Var98]*, and we can check the nonemptiness of $\mathcal{A}$ in time exponential in nk* [EJS93].

We use acronyms in $\{2, \varepsilon, 1\} \times \{A, N, D\} \times \{P, B, C, F\} \times \{T, W\}$ to denote the different types of automata. The first symbol stands for the type of movement: 2 for 2-way automata, ε for

advancing, and 1 for 1-way (we often omit the 1). The second symbol stands for the branching mode: A for alternating, N for nondeterministic, and D for deterministic. The third symbol stands for the type of acceptance: P for parity, B for Büchi, C for co-Büchi, and F for finite (i.e., automata that read finite words or trees), and the last symbol stands for the object the automaton is reading: T for trees and W for words. For example, a 2APT is a 2-way alternating parity tree automaton and an NBW is a 1-way nondeterministic Büchi word automaton.

The *membership problem* of an automaton $\mathcal{A}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$; or equivalently whether $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(A)$. It is not hard to see that the membership problem for a 2APT can be solved by a reduction to the emptiness problem. Formally we have the following.

Theorem 2.2.6 *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, and a regular tree $\langle \Upsilon^*, \tau \rangle$ we can check whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$ in time $(\|\tau\| nk)^{O(nk)}$.*

Proof: Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ be a 2APT and $\langle \Upsilon^*, \tau \rangle$ be a regular tree. Let the transducer inducing the labeling of τ be $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D, \eta, d_0, L \rangle$. According to Theorem 2.2.5, we construct a 1NPT $N = \langle \Sigma, S, \rho, s_0, \alpha \rangle$ that accepts the language of A.

Consider the 1NPT $N' = \langle \{a\}, D \times S, \rho', (d_0, s_0), \alpha' \rangle$ where $\rho'(d, s)$ is obtained from $\rho(s, L(d))$ by replacing every atom (v, s') by $(v, (\eta(d, v), s'))$ and α' is obtained from α by replacing every set F by the set $D \times F$. It follows that $\langle \Upsilon^*, \tau \rangle$ is accepted by $\mathcal{A}$ iff N' is not empty. The number of states of N' is $\|\tau\|(nk)^{O(nk)}$ and its index is $O(nk)$. □

2.2.5 Alternating Automata on Labeled Transition Graphs

Consider a labeled transition graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$. Let $\Delta = \{\varepsilon, \Box, \Diamond\}$. An alternating automaton on labeled transition graphs (*graph automaton*, for short) [JW95][5] is a tuple $\mathcal{S} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ, Q, q_0, and F are as in alternating two-way automata, and $\delta : Q \times \Sigma \to \mathcal{B}^+(\Delta \times Q)$ is the transition function. Intuitively, when $\mathcal{S}$ is in state q and it reads a state s of G, fulfilling an atom $\langle \Diamond, t \rangle$ (or $\Diamond t$, for short) requires $\mathcal{S}$ to send a copy in state t to some successor of s. Similarly, fulfilling an atom $\Box t$ requires $\mathcal{S}$ to send copies in state t to all the successors of s. Thus, like symmetric automata [DW99, Wil99], graph automata cannot distinguish between the various successors of a state and treat them in an existential or universal way.

Like runs of alternating two-way automata, a run of a graph automaton $\mathcal{S}$ over a labeled transition graph $G = \langle S, L, \rho, s_0 \rangle$ is a labeled tree in which every node is labeled by an element of $S \times Q$. A node labeled by (s, q), describes a copy of the automaton that is in the state q of $\mathcal{S}$ and reads the state s of G. Formally, a run is a Σ_r-labeled Γ-tree $\langle T_r, r \rangle$, where Γ is an arbitrary set of directions, $\Sigma_r = S \times Q$, and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (s_0, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (s, q)$ and $\delta(q, L(s)) = \theta$. Then there is a (possibly empty) set $S \subseteq \Delta \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, the following hold:
   - If $c = \varepsilon$, then there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s, q')$.

[5] The graph automata in [JW95] are different than these defined here, but this is only a technical difference.

- If $c = \Box$, then for every successor s' of s, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.
- If $c = \Diamond$, then there is a successor s' of s and $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.

A run $\langle T_r, r \rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. The graph G is accepted by $\mathcal{S}$ if there is an accepting run on it. We denote by $\mathcal{L}(\mathcal{S})$ the set of all graphs that $\mathcal{S}$ accepts. We denote by $\mathcal{S}^q = \langle \Sigma, Q, \delta, q, F \rangle$ the automaton $\mathcal{S}$ with q as its initial state.

We say that a labeled transition graph G satisfies a graph automaton $\mathcal{S}$, denoted $G \models \mathcal{S}$, if $\mathcal{S}$ accepts G. It is shown in [JW95] that graph automata are as expressive as μ-calculus. In particular, we have the following.

Theorem 2.2.7 [JW95] *Given a μ-calculus formula ψ, of length n and alternation depth k, we can construct a graph parity automaton $\mathcal{S}_\psi$ such that $L(\mathcal{S}_\psi)$ is exactly the set of graphs satisfying ψ. The automaton $\mathcal{S}_\psi$ has n states and index k.*

A graph automaton whose transitions are restricted to disjunctions over $\{\Diamond\} \times Q$ is in fact a nondeterministic automaton. We freely confuse between such graph automata with the Büchi acceptance condition and NBW. It is well known that every LTL formula can be translated into an NBW that accepts all traces that satisfy the formula. Formally, we have the following.

Theorem 2.2.8 [VW94] *For every LTL formula φ, we can construct an NBW N_φ with $2^{O(|\varphi|)}$ states such that $L(N_\varphi) = L(\varphi)$.*

2.2.6 Alternating Linear Space Turing Machines

An alternating Turing machine is $M = \langle \Gamma, S_u, S_e, \mapsto, s_0, F_{acc}, F_{rej} \rangle$, where the four sets of states ,S_u, S_e, F_{acc}, and F_{rej} are disjoint, and contain the universal, the existential, the accepting, and the rejecting states, respectively. We denote their union (the set of all states) by S. Our model of alternation prescribes that $\mapsto \subseteq S \times \Gamma \times S \times \Gamma \times \{L, R\}$ has a binary branching degree. When a universal or an existential state of M branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(s, a) \mapsto^l (s_l, b_l, \Delta_l)$ and $(s, a) \mapsto^r (s_r, b_r, \Delta_r)$ to indicate that when M is in state $s \in S_u \cup S_e$ reading input symbol a, it branches to the left with (s_l, b_l, Δ_l) and to the right with (s_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r.)

We consider here alternating linear-space Turing machines. Let $f : \mathbb{N} \to \mathbb{N}$ be the linear function such that M uses $f(n)$ cells in its working tape in order to process an input of length n. We encode a configuration of M by a string in $\{\sharp\} \cdot \Gamma^i \cdots (S \times \Gamma) \cdot \Gamma^{f(n)-i-1}$. That is, a configuration starts with the symbol $\sharp$, all its other letters are in Γ, except for one letter in $S \times \Gamma$. The meaning of such a configuration is that the j^{th} cell in the configuration, for $1 \leq j \leq f(n)$, is labeled γ_j, the reading head points at cell $i{+}1$, and M is in state s. For example, the initial configuration of M is $\sharp \cdot (s_0, b)b \cdots b$ (with $f(n){-}1$ occurrences of b's) where b stands for an empty cell. A configuration c' is a successor of configuration c if c' is a left or right successor of c. We can encode now a computation of M by a tree whose branches describe sequences of configurations of M. The computation is legal if a configuration and its successors satisfy the transition relation.

Note that though M has an existential (thus nondeterministic) mode, there is a single computation tree that describes all the possible choices of M. Each run of M corresponds to a pruning

of the computation tree in which all the universal configurations have both successors and all the existential configurations have at least one successor. The run is accepting if all the branches in the pruned tree reach an accepting configuration.

Theorem 2.2.9 [CKS81] *Deciding whether a linear space alternating Turing machine accepts the empty tape is EXPTIME-hard.*

2.3 Model-Checking Branching-Time Properties

In this section we present an automata-theoretic approach solution to model-checking branching-time properties of pushdown and prefix-recognizable graphs. We start by demonstrating our technique on model checking of pushdown systems. Then we show how to extend it to prefix-recognizable systems. Consider a rewrite system $R = \langle \Sigma,\ V,\ Q,\ L,\ T,\ q_0,\ x_0 \rangle$ and let $G_R = \langle \Sigma,\ Q \times V^*,\ L,\ \rho_R,\ (q_0, x_0) \rangle$ be its induced graph. recall that a configuration of G_R is a pair $(q, x) \in Q \times V^*$. Thus, the store x corresponds to a node in the full infinite V-tree. An automaton that reads the tree V^* can memorize in its state space the state component of the configuration and refer to the location of its reading head in V^* as the store. We would like the automaton to "know" the location of its reading head in V^*. A straightforward way to do so is to label a node $x \in V^*$ by x. This, however, involves an infinite alphabet, and results in trees that are not regular.

We show that labeling every node in V^* by its direction is sufficiently informative to provide the 2-way automaton with the information it needs in order to simulate transitions of the rewrite system. Thus, if R is a pushdown system and we are at node $A \cdot y$ of the V-tree (with state q memorized), an application of the transition $\langle q, A, x, q' \rangle$ takes us to node $x \cdot y$ (with state q' memorized). If R is a prefix-recognizable system and we are at node y of the V-tree (with state q memorized), an application of the transition $\langle q, \alpha, \beta, \gamma, q' \rangle$ takes us to node xz (with state q' memorized) where $x \in \gamma$, $z \in \beta$, and $y = z'z$ for some $z' \in \alpha$. Technically, this means that we first move up the tree, and then move down. Such a navigation through the V-tree can be easily performed by two-way automata.

2.3.1 Pushdown Graphs

We present our solution for pushdown graphs in details. Let $\langle V^*, \tau_V \rangle$ be the V-labeled V-tree such that for every $x \in V^*$ we have $\tau_V(x) = dir(x)$ ($\langle V^*, \tau_V \rangle$ is the exhaustive V-labeled V-tree). Note that $\langle V^*, \tau_V \rangle$ is a regular tree of size $|V| + 1$. We construct a 2APT $\mathcal{A}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{A}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{A}$ consults its current state and the label of the node it reads (note that $dir(x)$ is the first letter of x). Formally, we have the following.

Theorem 2.3.1 *Given a pushdown system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma,\ W,\ \delta,\ w_0,\ F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\bot\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $G_{\mathcal{R}}$ satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\|)$ states, and has the same index as $\mathcal{S}$.*

Proof: We define $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ as follows.

- $P = (W \times Q \times \mathit{heads}(T))$, where $\mathit{heads}(T) \subseteq V^*$ is the set of all prefixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{A}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that G_R with initial state $(q, y \cdot x)$ is accepted by $\mathcal{S}^s$. In particular, when $y = \varepsilon$, then G_R with initial state (q, x) (the node currently being visited) needs to be accepted by $\mathcal{S}^w$. States of the form $\langle w, q, \varepsilon \rangle$ are called *action states*. From these states $\mathcal{A}$ consults δ and T in order to impose new requirements on the exhaustive V-tree. States of the form $\langle w, q, y \rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{A}$ only navigates downwards y to reach new action states.

- In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(\mathit{ext}(V) \times P)$, we first define the function $\mathit{apply}_T : \Delta \times W \times Q \times V \to \mathcal{B}^+(\mathit{ext}(V) \times P)$. Intuitively, apply_T transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words of the form $A \cdot V^*$. For $c \in \Delta$, $w \in W$, $q \in Q$, and $A \in V$ we define

$$\mathit{apply}_T(c, w, q, A) = \begin{cases} \langle \varepsilon, (w, q, \varepsilon) \rangle & \text{If } c = \varepsilon \\ \bigwedge_{\langle q, A, y, q' \rangle \in T} \langle \uparrow, (w, q', y) \rangle & \text{If } c = \Box \\ \bigvee_{\langle q, A, y, q' \rangle \in T} \langle \uparrow, (w, q', y) \rangle & \text{If } c = \Diamond \end{cases}$$

  Note that T may contain no tuples in $\{q\} \times \{A\} \times V^* \times Q$ (that is, the transition relation of G_R may not be total). In particular, this happens when $A = \bot$ (that is, for every state $q \in Q$ the configuration (q, ε) of G_R has no successors). Then, we take empty conjunctions as **true**, and take empty disjunctions as **false**.

  In order to understand the function apply_T, consider the case $c = \Box$. When $\mathcal{S}$ reads the configuration $(q, A \cdot x)$ of the input graph, fulfilling the atom $\Box w$ requires $\mathcal{S}$ to send copies in state w to all the successors of $(q, A \cdot x)$. The automaton $\mathcal{A}$ then sends to the node x copies that check whether all the configuration $(q', y \cdot x)$, with $\rho_R((q, A \cdot x), (q', y \cdot x))$, are accepted by $\mathcal{S}$ with initial state w.

  Now, for a formula $\theta \in \mathcal{B}^+(\Delta \times S)$, the formula $\mathit{apply}_T(\theta, q, A) \in \mathcal{B}^+(\mathit{ext}(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w \rangle$ by the atom $\mathit{apply}_R(c, w, q, A)$. We can now define η for all $A \in V \cup \{\bot\}$ as follows.

  - $\eta(\langle w, q, \varepsilon \rangle, A) = \mathit{apply}_T(\delta(w, L(q, A)), q, A)$.
  - $\eta(\langle w, q, y \cdot B \rangle, A) = (B, \langle w, q, y \rangle)$.

  Thus, in action states, $\mathcal{A}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to δ. In navigation states, $\mathcal{A}$ needs to go downwards $y \cdot B$, so it continues in direction B.

- $p_0 = \langle w_0, q_0, x_0 \rangle$. Thus, in its initial state $\mathcal{A}$ checks that G_R with initial configuration (q_0, x_0) is accepted by $\mathcal{S}$ with initial state w_0.

- α is obtained from F by replacing each set F_i by the set $S \times F_i \times \mathit{heads}(T)$.

We show that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \models \mathcal{S}$. Assume that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $\langle T, r \rangle$ of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subtree of nodes labeled by action states. That is, consider the following tree $\langle T', r' \rangle$ defined by induction. We know that $r(\varepsilon) = (\varepsilon, (w_0, q_0, x_0))$. It follows that there exists a unique minimal (according to the inverse lexicographic order on the nodes of T) node $y \in T$ labeled by an action state. In our case,

$r(y) = (x_0, (w_0, q_0, \varepsilon))$. We add ε to T' and label it $r'(\varepsilon) = ((q_0, x_0), w_0)$. Consider a node z' in T' labeled $r'(z') = ((q, x), w)$. By the construction of $\langle T', r' \rangle$ there exists $z \in T$ such that $r(z) = (x, (w, q, \epsilon))$. Let $\{z_1 \cdot z, \ldots, z_k \cdot z\}$ be the set of minimal nodes in T such that $z_i \cdot z$ is labeled by an action state, $r(z_i \cdot z) = (x_i, (w_i, q_i, \varepsilon))$. We add k successors $a_1 z', \ldots a_k z'$ to z' in T' and set $r'(a_i z') = ((q_i, x_i), w_i)$. By the definition of η, the tree $\langle T', r' \rangle$ is a valid run tree of $\mathcal{S}$ on G_R. Consider an infinite path π' in $\langle T', r' \rangle$. The labels of nodes in π' identify a unique path π in $\langle T, r \rangle$. It follows that the minimal rank appearing infinitely often along π' is the minimal rank appearing infinitely often along π. Hence, $\langle T', r' \rangle$ is accepting and $\mathcal{S}$ accepts G_R.

Assume now that $G_R \models \mathcal{S}$. Then, there exists an accepting run tree $\langle T', r' \rangle$ of $\mathcal{S}$ on G_R. The tree $\langle T', r' \rangle$ serves as the action state skeleton to an accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. A node $z \in T'$ labeled by $((q, x), w)$ corresponds to a copy of $\mathcal{A}$ in state (w, q, ε) reading node x of $\langle V^*, \tau_V \rangle$. It is not hard to extend this skeleton into a valid and accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. □

Pushdown systems can be viewed as a special case of prefix-recognizable systems. In the next subsection we describe how to extend the construction described above to prefix-recognizable graphs, and we also analyze the complexity of the model-checking algorithm that follows for the two types of systems.

2.3.2 Prefix-Recognizable Graphs

In this section we extend the construction described in Subsection 2.3.1 to prefix-recognizable systems. The idea is similar: two-way automata can navigate through the full V-tree and simulate transitions in a rewrite graph by a chain of transitions in the tree. While in pushdown systems the application of rewrite rules involved one move up the tree and then a chain of moves down, here things are a bit more involved. In order to apply a rewrite rule $\langle q, \alpha, \beta, \gamma, q' \rangle$, the automaton has to move upwards along a word in α, check that the remaining word leading to the root is in β, and move downwards along a word in γ. As we explain below, $\mathcal{A}$ does so by simulating automata for the regular expressions participating in T.

Theorem 2.3.2 *Given a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\bot\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff G_R satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\|)$ states, and its index is the index of $\mathcal{S}$ plus 1.*

Proof: As in the case of pushdown systems, $\mathcal{A}$ uses the labels on $\langle V^*, \tau_V \rangle$ in order to learn the location in V^* that each node corresponds to. As there, $\mathcal{A}$ applies to the transition function δ of $\mathcal{S}$ the rewrite rules of R. Here, however, the application of the rewrite rules on atoms of the form $\Diamond w$ and $\Box w$ is more involved, and we describe it below. Assume that $\mathcal{A}$ wants to check whether $\mathcal{S}^w$ accepts $G_R^{(q,x)}$, and it wants to proceed with an atom $\Diamond w'$ in $\delta(w, L(q, x))$. The automaton $\mathcal{A}$ needs to check whether $\mathcal{S}^{w'}$ accepts $G_R^{(q',y)}$ for some configuration (q', y) reachable from (q, x) by applying a rewrite rule. That is, a configuration (q', y) for which there is $\langle q, \alpha, \beta, \gamma, q' \rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_\alpha$, z is accepted by $\mathcal{U}_\beta$, and y' accepted by $\mathcal{U}_\gamma$. The way $\mathcal{A}$ detects such a state y is the following. From the node x, the automaton $\mathcal{A}$ simulates the automaton $\mathcal{U}_\alpha$ upwards (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_\alpha$ on the word it reads as it proceeds on direction $\uparrow$ from x towards the root of the V-tree). Suppose that

on its way up to the root, $\mathcal{A}$ encounters a state in F_α as it reads the node $z \in V^*$. This means that the word read so far is in α, and can serve as the prefix x' above. If this is indeed the case, then it is left to check that the word z is accepted by $\mathcal{U}_\beta$, and that there is a state that is obtained from z by prefixing it with a word $y' \in \gamma$ that is accepted by $\mathcal{S}^{w'}$. To check the first condition, $\mathcal{A}$ sends a copy in direction $\uparrow$ that simulates a run of $\mathcal{U}_\beta$, hoping to reach a state in F_β as it reaches the root (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_\beta$ on the word it reads as it proceeds from z up to the root of $\langle V^*, \tau_V \rangle$). To check the second condition, $\mathcal{A}$ simulates the automaton $\mathcal{U}_\gamma$ downwards starting from F_γ. A node $y' \cdot z \in V^*$ that $\mathcal{A}$ reads as it encounters q^0_γ can serve as the state y we are after. The case for an atom $\Box w'$ is similar, only that here $\mathcal{A}$ needs to check whether $\mathcal{S}^{w'}$ accepts $G_R^{(q',y)}$ for all configurations (q', y) reachable from (q, x) by applying a rewrite rule, and thus the choices made by $\mathcal{A}$ for guessing the partition $x' \cdot z$ of x and the prefix y' of y are now treated dually. More formally, we have the following.

We define $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ as follows.

- $P = P_1 \cup P_2$ where $P_1 = \{\exists, \forall\} \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma)$ and $P_2 = \{\exists, \forall\} \times T \times Q_\beta$. States in P_1 serve to simulate automata for α and γ regular expressions and states in P_2 serve to simulate automata for β regular expressions. A state marked by $\exists$ participates in the simulation of a $\Diamond s$ atom of $\mathcal{S}$, and a state marked by $\forall$ participates in the simulation of a $\Box s$ atom of $\mathcal{S}$. A state in P_1 marked by the transition $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ and a state $s \in Q_{\alpha_i}$ participates in the simulation of a run of $\mathcal{U}_{\alpha_i}$. When $s \in F_{\alpha_i}$ (recall that states in F_{α_i} have no outgoing transitions) $\mathcal{A}$ spawns a copy (in a state in P_2) that checks that the suffix is in β_i and continues to simulate $\mathcal{U}_{\gamma_i}$. A state in P_1 marked by the transition $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ and a state $s \in Q_{\gamma_i}$ participates in the simulation of a run of U_{γ_i}. When $s = q_o^{\gamma_i}$ (recall that the initial state $q^0_{\gamma_i}$ has no incoming transitions) the state is an action state, and $\mathcal{A}$ consults δ and T in order to impose new restrictions on $\langle V^*, \tau_V \rangle$. [6]

- In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(ext(V) \times P)$, we first define the function $apply_T : \Delta \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma) \to \mathcal{B}^+(ext(V) \times P)$. Intuitively, $apply_T$ transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words from V^*. For $c \in \Delta$, $w \in W$, $q \in Q$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$, and $s = q^0_{\gamma_i}$ we define

$$apply_T(c, w, q, t_i, s) = \begin{cases} \langle \varepsilon, (\exists, w, q, t_i, s) \rangle & \text{If } c = \varepsilon \\ \bigwedge_{t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle \in T} (\varepsilon, (\forall, w, q', t_{i'}, q^0_{\alpha_{i'}})) & \text{If } c = \Box \\ \bigvee_{t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle \in T} (\varepsilon, (\exists, w, q', t_{i'}, q^0_{\gamma_{i'}})) & \text{If } c = \Diamond \end{cases}$$

  In order to understand the function $apply_T$, consider the case $c = \Box$. When $\mathcal{S}$ reads the configuration (q, x) of the input graph, fulfilling the atom $\Box w$ requires $\mathcal{S}$ to send copies in state w to all the successors of (q, x). The automaton $\mathcal{A}$ then sends copies that check whether all the configurations (q', y') with $\rho_R((q, x), (q', y'))$ are accepted by $\mathcal{S}$ with initial state w.

  Now, for a formula $\theta \in \mathcal{B}^+(\Delta \times W)$, the formula $apply_T(\theta, q, t_i, s) \in \mathcal{B}^+(ext(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w \rangle$ by the atom $apply_T(c, w, q, t_i, s)$. We can now

[6] Note that a straightforward representation of P results in $O(|W| \cdot |Q| \cdot |T| \cdot \|T\|)$ states. Since, however, the states of the automata for the regular expressions are disjoint, we can assume that the rewrite rule in T that each automaton corresponds to is uniquely defined from it.

define η for all $w \in W$, $q \in Q$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \in T$, $s \in Q_{\alpha_i} \cup Q_{\gamma_i}$, and $A \in V \cup \{\bot\}$ as follows.

$$\eta((\exists, w, q, t_i, s), A) = \begin{cases} apply_T(\delta(w, L(q, A)), q, t_i, s) & s = q^0_{\gamma_i} \\ \bigvee_{B \in V} \bigvee_{s \in \eta_{\gamma_i}(s', B)} (B, (\exists, w, q, t_i, s')) & s \in Q_{\gamma_i} \setminus \{q^0_{\gamma_i}\} \\ \bigvee_{s' \in \eta_{\alpha_i}(s, A)} (\uparrow, (\exists, w, q, t_i, s')) & s \in Q_{\alpha_i} \setminus F_{\alpha_i} \\ (\varepsilon, (\exists, t_i, q^0_{\beta_i})) \wedge \left(\bigvee_{s' \in F_{\gamma_i}} (\varepsilon, (\exists, w, q, t_i, s'))\right) & s \in F_{\alpha_i} \end{cases}$$

$$\eta((\forall, w, q, t_i, s), A) = \begin{cases} apply_T(\delta(w, L(q, A)), q, t_i, s) & s = q^0_{\gamma_i} \\ \bigwedge_{B \in V} \bigwedge_{s \in \eta_{\gamma_i}(s', B)} (B, (\forall, w, q, t_i, s')) & s \in Q_{\gamma_i} \setminus \{q^0_{\gamma_i}\} \\ \bigwedge_{s' \in \eta_{\alpha_i}(s, A)} (\uparrow, (\forall, w, q, t_i, s')) & s \in Q_{\alpha_i} \setminus F_{\alpha_i} \\ (\varepsilon, (\forall, t_i, q^0_{\beta_i})) \vee \left(\bigwedge_{s' \in F_{\gamma_i}} (\varepsilon, (\forall, w, q, t_i, s'))\right) & t \in F_{\alpha_i} \end{cases}$$

Thus, when $s \in Q_\alpha$ the 2APT $\mathcal{A}$ either chooses a successor s' of s and goes up the tree or in case s is an accepting state of $\mathcal{U}_{\alpha_i}$, it spawns a copy that checks that the suffix is in β_i and moves to a final state of $\mathcal{U}_{\gamma_i}$.

When $s \in Q_\gamma$ the 2APT $\mathcal{A}$ either chooses a direction B and chooses a predecessor s' of s or in case that $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{A}$ uses the transition δ to impose new restrictions on $\langle V^*, \tau_V\rangle$.

We define η for all $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$, $s \in Q_{\beta_i}$, and $A \in V \cup \{\bot\}$ as follows.

$$\eta((\exists, t_i, s), A) = \begin{cases} \bigvee_{s' \in \eta_{\beta_i}(s, A)} (\uparrow, (\exists, t_i, s')) & A \neq \bot \\ \textbf{true} & s \in F_{\beta_i} \text{ and } A = \bot \\ \textbf{false} & s \notin F_{\beta_i} \text{ and } A = \bot \end{cases}$$

$$\eta((\forall, t_i, s), A) = \begin{cases} \bigwedge_{s' \in \eta_{\beta_i}(s, A)} (\uparrow, (\forall, t_i, s')) & A \neq \bot \\ \textbf{false} & s \in F_{\beta_i} \text{ and } A = \bot \\ \textbf{true} & s \notin F_{\beta_i} \text{ and } A = \bot \end{cases}$$

If $s \in Q_\beta$, then in existential mode, the automaton $\mathcal{A}$ makes sure that the suffix is in β and in universal mode it makes sure that the suffix is not in β.

- $p_0 = \langle \exists, w_0, q_0, t, x_0\rangle$. Thus, in its initial state $\mathcal{A}$ starts a simulation (backward) of the automaton that accepts the unique word x_0. It follows that $\mathcal{A}$ checks that G_R with initial configuration (q_0, x_0) is accepted by $\mathcal{S}$ with initial state w_0.

- Let $F_\gamma = \bigcup_{t_i \in T} F_{\gamma_i}$. The acceptance condition α is obtained from F by replacing each set F_i by the set $\{\exists, \forall\} \times F_i \times Q \times T \times F_\gamma$. We add to α a maximal odd set and include all the states in $\{\exists\} \times W \times Q \times T \times (Q_\gamma \setminus F_\gamma)$ in this set. We add to α a maximal even set and include all the states in $\{\forall\} \times W \times Q \times T \times (Q_\gamma \setminus F_\gamma)$ in this set[7]. The states in $\{\exists, \forall\} \times W \times Q \times T \times Q_\alpha$ and P_2 are added to the maximal set (notice that states marked by a state in Q_α appear in finite sequences and states in P_2 appear only in suffixes of finite paths in the run tree).

  Thus, in a path that visits infinitely many action states, the action states define it as accepting or not accepting. A path that visits finitely many action states is either finite or ends in an infinite sequence of Q_γ labeled states. If these states are existential, then the path is rejecting. If these states are universal, then the path is accepting.

[7]Notice, that if the maximal set in F is even then we only add to α a maximal odd set. Dually, if the maximal set in F is odd then we add to α a maximal even set.

We show that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \models \mathcal{S}$. Assume that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $\langle T, r \rangle$ of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subtree of nodes labeled by action states. Denote this tree by $\langle T', r' \rangle$. By the definition of δ, the tree $\langle T', r' \rangle$ is a valid run tree of $\mathcal{S}$ on G_R. Consider an infinite path π' in $\langle T', r' \rangle$. The labels of nodes in π' identify a unique path π in $\langle T, r \rangle$. As π' is infinite, it follows that π visits infinitely many action states. As all navigation states are added to the maximal ranks the minimal rank visited along π must be equal to the minimal rank visited along π'. Hence, $\langle T', r' \rangle$ is accepting and $\mathcal{S}$ accepts G_R.

Assume now that $G_R \models \mathcal{S}$. Then, there exists an accepting run tree $\langle T', r' \rangle$ of $\mathcal{S}$ on G_R. The tree $\langle T', r' \rangle$ serves as the action state skeleton to an accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. A node $z \in T'$ labeled by $((q, x), w)$ corresponds to a copy of $\mathcal{A}$ in state (d, w, q, t, s) reading node x of $\langle V^*, \tau_V \rangle$ for some $d \in \{\exists, \forall\}$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$ and $s = q^0_{\gamma_i}$. In order to extend this skeleton into a valid and accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$ we have to complete the runs of the automata for the different regular expressions appearing in T. □

The constructions described in Theorems 2.3.1 and 2.3.2 reduce the model-checking problem to the membership problem of $\langle V^*, \tau_V \rangle$ in the language of a 2APT. By Theorem 2.2.6, we then have the following.

Theorem 2.3.3 *The model-checking problem for a pushdown or a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$ and k is the index of $\mathcal{S}$.*

Together with Theorem 2.2.7, we can conclude with an EXPTIME bound also for the model-checking problem of μ-calculus formulas matching the lower bound in [Wal96]. Note that the fact the same complexity bound holds for both pushdown and prefix-recognizable rewrite systems stems from the different definition of $\|T\|$ in the two cases.

2.4 Path Automata on Trees

We would like to enhance the approach developed in Section 2.3 to linear time properties. The solution to μ-calculus model checking is exponential in both the system and the specification and it is EXPTIME-complete [Wal96]. On the other hand, model-checking linear-time specifications is polynomial in the system [BEM97]. As we discuss below, both the emptiness and membership problems for 2APT are EXPTIME-complete. While 2APT can reason about many computation paths simultaneously, in linear-time model-checking we need to reason about a single path that does not satisfy a specification. It follows, that the extra power of 2APT comes at a price we cannot pay. In this section we introduce *path automata* and study them. In Section 2.5 we show that path automata give us the necessary tool in order to reason about linear specifications.

Path automata resemble *tree walking automata*. These are automata that read finite trees and expect the nodes of the tree to be labeled by the direction and by the set of successors of the node. Tree walking automata are used in XML queries. We refer the reader to [EHvB99, Nev02].

2.4.1 Definition

Path automata on trees are a hybrid of nondeterministic word automata and nondeterministic tree automata: they run on trees but have linear runs. Here we describe *two-way* nondeterministic Büchi path automata.

A *two-way nondeterministic Büchi path automaton* (2NBP, for short) on Σ-labeled Υ-trees is in fact a 2ABT whose transitions are restricted to disjunctions. Formally, $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$, where Σ, P, p_0, and F are as in an NBW, and $\delta : P \times \Sigma \to 2^{(ext(\Upsilon) \times P)}$ is the transition function. A path automaton that is in state p and reads the node $x \in T$ chooses a pair $(d, p') \in \delta(p, \tau(x))$, and then follows direction d and moves to state p'. It follows that a *run* of a 2NBP $\mathcal{P}$ on a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a sequence of pairs $r = (x_0, p_0), (x_1, p_1), \ldots$ where for all $i \geq 0$, $x_i \in \Upsilon^*$ is a node of the tree and $p_i \in P$ is a state. The pair (x, p) describes a copy of the automaton that reads the node x of Υ^* and is in the state p. Note that many pairs in r may correspond to the same node of Υ^*; Thus, $\mathcal{S}$ may visit a node several times. The run has to satisfy the transition function. Formally, $(x_0, p_0) = (\varepsilon, q_0)$ and for all $i \geq 0$ there is $d \in ext(\Upsilon)$ such that $(d, p_{i+1}) \in \delta(p_i, \tau(x_i))$ and

- If $\Delta \in \Upsilon$, then $x_{i+1} = \Delta \cdot x_i$.
- If $\Delta = \varepsilon$, then $x_{i+1} = x_i$.
- If $\Delta = \uparrow$, then $x_i = v \cdot z$, for some $v \in \Upsilon$ and $z \in \Upsilon^*$, and $x_{i+1} = z$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $d = \uparrow$, we require that $x_i \neq \varepsilon$. A run r is *accepting* if it visits $\Upsilon^* \times F$ infinitely often. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{P})$ the set of all Σ-labeled trees that $\mathcal{P}$ accepts. The automaton $\mathcal{P}$ is *nonempty* iff $\mathcal{L}(\mathcal{P}) \neq \emptyset$. We measure the size of a 2NBP by two parameters, the number of states and the size, $|\delta| = \Sigma_{p \in P} \Sigma_{a \in \Sigma} |\delta(s, a)|$, of the transition function.

Readers familiar with tree automata know that the run of a tree automaton starts in a single copy of the automaton reading the root of the tree, and then the copy splits to the successors of the root and so on, thus the run simultaneously follows many paths in the input tree. In contrast, a path automaton has a single copy at all times. It starts from the root and it always chooses a single direction to go to. In two-way path automata, the direction may be "up", so the automaton can read many paths of the tree, but it cannot read them simultaneously.

The fact that a 2NBP has a single copy influences its expressive power and the complexity of its nonemptiness and membership problems. We now turn to study these issues.

2.4.2 Expressiveness

One-way nondeterministic path automata can read a single path of the tree, so it is easy to see that they accept exactly all languages $\mathcal{T}$ of trees such that there is an ω-regular language L of words and $\mathcal{T}$ contains exactly all trees that have a path labeled by a word in L. For two-way path automata, the expressive power is less clear, as by going up and down the tree, the automaton can traverse several paths. Still, a path automaton cannot traverse all the nodes of the tree. To see that, we prove that a 2NBP cannot recognize even very simple properties that refer to all the branches of the tree (*universal* properties for short).

Theorem 2.4.1 *There are no 2NBP $\mathcal{P}_1$ and $\mathcal{P}_2$ over the alphabet $\{0, 1\}$ such that $L(\mathcal{P}_1) = L_1$ and $L(\mathcal{P}_2) = L_2$ where $|\Upsilon| > 1$ and*

- $L_1 = \{\langle \Upsilon^*, \tau \rangle : \tau(x) = 0$ *for all* $x \in T\}$.

- $L_2 = \{\langle \Upsilon^*, \tau \rangle$: *for every path* $\pi \subseteq T$, *there is* $x \in \pi$ *with* $\tau(x) = 0\}$.

Proof: Suppose that there exists a 2NBP $\mathcal{P}_1$ that accepts L_1. Let $T = \langle \Upsilon^*, \tau \rangle \in L_1$ be some tree accepted by $\mathcal{P}_1$. There exists an accepting run $r = (x_0, p_0), (x_1, p_1), \ldots$ of $\mathcal{P}_1$ on T. It is either the case that r visits some node in Υ^* infinitely often or not.

- Suppose that there exists a node $x \in \Upsilon^*$ visited infinitely often by r. There must exist $i < j$ such that $x_i = x_j = x$, $p_i = p_j$, and there exists $i \leq k < j$ such that $p_k \in F$. Consider the run $r' = (x_0, p_0), \ldots, (x_{i-1}, p_{i-1})\,((x_i, p_i), \ldots, (x_{j-1}, p_{j-1}))^\omega$. Clearly, it is a valid and accepting run of $\mathcal{P}_1$ on T. However, r' visits only a finite number of nodes in T. Let $W = \{x_i | x_i \text{ visited by } r'\}$. It is quite clear that the same run r' is an accepting run of $\mathcal{P}_1$ on the tree $\langle \Upsilon, \tau' \rangle$ such that $\tau'(x) = \tau(x)$ for $x \in W$ and $\tau'(x) = 1$ for $x \notin W$. Clearly, $\langle \Upsilon^*, \tau' \rangle \notin L_1$.
- Suppose that every node $x \in \Upsilon^*$ is visited only a finite number of times. Let (x_i, p_i) be the last visit of r to the root. It must be the case that $x_{i+1} = \upsilon$ for some $\upsilon \in \Upsilon$. Let $\upsilon' \neq \upsilon$ be a different element in Υ. Let $W = \{x_{i'} \in \Upsilon^* \cdot \upsilon' \mid x_{i'} \text{ visited by } r\}$ be the set of nodes in the subtree of υ' visited by r. Clearly, W is finite and we proceed as above.

The proof for the case of $\mathcal{P}_2$ and L_2 is similar. □

There are, however, universal properties that a 2NBP can recognize. Consider a language $L \subseteq \Sigma^\omega$ of infinite words over the alphabet Σ. A finite word $x \in \Sigma^*$ is a *bad prefix* for L iff for all $y \in \Sigma^\omega$, we have $x \cdot y \notin L$. Thus, a bad prefix is a finite word that cannot be extended to an infinite word in L. A language L is a *safety* language iff every $w \notin L$ has a finite bad prefix. A language $L \subseteq \Sigma^\omega$ is *clopen* if both L and its complement are safety languages, or, equivalently, L corresponds to a set that is both closed and open in Cantor space. It is known that a clopen language is bounded: there is an integer k such that after reading a prefix of length k of a word $w \in \Sigma^\omega$, one can determine whether w is in L [KV01]. A 2NBP can then traverse all the paths of the input tree up to level k (given L, its bound k can be calculated), hence the following theorem.

Theorem 2.4.2 *Let* $L \subseteq \Sigma^\omega$ *be a clopen language. There is a 2NBP* $\mathcal{P}$ *such that* $L(\mathcal{P}) = \{\langle \Upsilon^*, \tau \rangle$: *for all paths* $\pi \subseteq \Upsilon^*$, *we have* $\tau(\pi) \in L\}$.

Proof: Let k be the bound of L and $\Upsilon = \{\upsilon_1, \ldots, \upsilon_m\}$. Consider, $w = w_0, \ldots, w_r \in \Upsilon^*$. Let i be the maximal index such that $w_i \neq \upsilon_m$. We set $succ(w) = w_0, \ldots w_{i-1}, w_i', w_{i+1}, \ldots, w_r$ where if $w_i = \upsilon_j$ then $w_i' = \upsilon_{j+1}$. That is, if we take $w = (\upsilon_1)^k$ then by using the *succ* function we pass on all elements in Υ^k according to the lexicographic order (induced by $\upsilon_1 < \upsilon_2 < \ldots < \upsilon_m$). Let $\mathcal{N} = \langle \Sigma, N, \delta, n_0, F \rangle$ be an NBW accepting L. According to [KV01], $\mathcal{N}$ is cycle-free and has a unique accepting sink state. Formally, $\mathcal{N}$ has an accepting state n_{acc} such that for every $\sigma \in \Sigma$ we have $\delta(n_{acc}, \sigma) = \{n_{acc}\}$ and for every run $r = n_0, n_1, \ldots$ and every $i < j$ either $n_i \neq n_j$ or $n_i = n_{acc}$.

We construct a 2NBP that scans all the paths in Υ^k according to the order induced by using *succ*. The 2NBP scans a path and simulates $\mathcal{N}$ on this path. Once our 2NBP ensures that this path is accepted by $\mathcal{N}$ it proceeds to the next path. Consider the following 2NBP $\mathcal{P} = \langle \Sigma, Q, \eta, q_0, \{q_{acc}\} \rangle$ where

- $Q = (\{u, d\} \times \Upsilon^k \times [k] \times N) \cup \{q_{acc}\}$. A state consists of 4 components. The symbols u and d are acronyms for *up* and *down*. A state marked by d means that the 2NBP is going down the tree while scanning a path. A state marked by u means that the 2NBP is going up towards the root where it starts scanning the next path. The word $w \in \Upsilon^k$ is the current explored path. The number $i \in [k]$ denotes the location in the path w. The state $n \in N$ denotes the current state of the automaton $\mathcal{N}$.

- For every state $q \in Q$ and letter $\sigma \in \Sigma$, the transition function $\eta : Q \times \Sigma \to 2^{ext(\Upsilon) \times Q}$ is defined as follows:

$$\eta((d, w, i, n), \sigma) = \begin{cases} \{(w_{i+1}, (d, w, i{+}1, n')) \mid n' \in \delta(n, \sigma)\} & i \neq k \\ \emptyset & i = k \text{ and } n \neq n_{acc} \\ \{(\varepsilon, (u, succ(w), i, n))\} & i = k, n = n_{acc}, \text{ and } w \neq (\upsilon_m)^k \\ \{(\varepsilon, q_{acc})\} & i = k, n = n_{acc}, \text{ and } w = (\upsilon_m)^k \end{cases}$$

$$\eta((u, w, i, n), \sigma) = \begin{cases} \{(\uparrow, (u, w, i-1, n))\} & i \neq 0 \\ \{(\varepsilon, (d, w, 0, n_0))\} & i = 0 \end{cases}$$

$$\eta(q_{acc}, \sigma) = \{(\varepsilon, q_{acc})\}$$

  Intuitively, in d-states the automaton goes in the direction dictated by w and simulates $\mathcal{N}$ on the labeling of the path w. Once the path w is explored, if the $\mathcal{N}$ component is not in n_{acc} this means the run of $\mathcal{N}$ on w failed and the run is terminated. If the $\mathcal{N}$ component reaches n_{acc} this means that the run of $\mathcal{N}$ on w succeeded and the 2NBP proceeds to a u-state with $succ(w)$. If $succ(w)$ does not exist (i.e., $w = (\upsilon_m)^k$) the 2NBP accepts. In u-states the 2NBP goes up towards the root; when it reaches the root it initiates a run of $\mathcal{N}$ on the word w.

- $q_0 = (d, (\upsilon_1)^k, 0, n_0)$. Thus, in the initial state, $\mathcal{P}$ starts to simulate $\mathcal{N}$ on the first path $(\upsilon_1)^k$.

It is quite simple to see that every tree in L is accepted by $\mathcal{P}$ and that every tree accepted by $\mathcal{P}$ is in L. □

The question whether walking tree automata accept all regular tree languages is an open problem [Nev02]. Recently, it was shown that deterministic walking tree automata are less expressive than nondeterministic walking tree automata [BC04]. That is, there exist languages recognized by nondeterministic walking tree automata that cannot be recognized by deterministic walking tree automata. Using standard techniques to generalize results about automata over finite objects to automata over infinite objects we can show that 2DBP are less expressive than 2NBP. Similarly, the algorithms described in the next subsection can be modified to handle the respective problems for walking tree automata.

2.4.3 Decision Problems

Given a 2NBP $\mathcal{S}$, the *emptiness problem* is to determine whether $\mathcal{S}$ accepts some tree, or equivalently whether $\mathcal{L}(\mathcal{S}) = \emptyset$. The *membership problem* of $\mathcal{S}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{S}$ accepts $\langle \Upsilon^*, \tau \rangle$, or equivalently $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{S})$. The fact that 2NBP cannot spawn new copies makes them very similar to word automata. Thus, the membership problem for 2NBP can be reduced to the emptiness problem of εABW over a 1-letter alphabet (cf. [KVW00]). The reduction

yields a polynomial time algorithm for solving the membership problem. In contrast, the emptiness problem of 2NBP is EXPTIME-complete.

We show a reduction from the membership problem of 2NBP to the emptiness problem of εABW with a 1-letter alphabet. The reduction is a generalization of a construction that translates 2NBW to εABW [PV03]. The emptiness of εABW with a 1-letter alphabet is solvable in quadratic time and linear space [KVW00]. We show that in our case the membership problem of a 2NBP is solved in cubic time and quadratic space in the size of the original 2NBP. Formally, we have the following.

Theorem 2.4.3 *Consider a 2NBP $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$. The membership problem of the regular tree $\langle \Upsilon^*, \tau \rangle$ in the language of $\mathcal{S}$ is solvable in time $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$ and space $O(|P|^2 \cdot \|\tau\|)$.*

Proof: We construct an εABW on 1-letter alphabet $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ such that $L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$. The εABW $\mathcal{A}$ has $O(|P|^2 \cdot \|\tau\|)$ states and the size of its transition function is $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$. We use the fact that $\mathcal{A}$ is advancing and remember the state of the transducer that gives the label to the current node in the tree $\langle \Upsilon^*, \tau \rangle$ as part of the finite control of $\mathcal{A}$. For better intuition of the construction we refer the reader to [PV03].

Let $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D_\tau, \rho_\tau, d_0^\tau, L_\tau \rangle$ be the transducer that generates the labels of τ. For a word $w \in \Upsilon^*$ we denote by $\rho_\tau(w)$ the unique state that $\mathcal{D}_\tau$ gets to after reading w. We construct the εABW $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ as follows.

- $Q = (P \cup (P \times P)) \times D_\tau \times \{\bot, \top\}$. States in $P \times D_\tau \times \{\bot, \top\}$, which hold a single state from P, are called *singleton states*. Similarly, we call states in $P \times P \times D_\tau \times \{\bot, \top\}$ *pair states*.
- $q_0 = (p_0, d_0^\tau, \bot)$.
- $\alpha = (F \times D_\tau \times \{\bot\}) \cup (P \times D_\tau \times \{\top\})$.

In order to define the transition function we have the following definitions. Two functions $f_\alpha : P \times P \to \{\bot, \top\}$ where $\alpha \in \{\bot, \top\}$, and for every state $p \in P$ and alphabet letter $\sigma \in \Sigma$ the set C_p^σ is the set of states from which p is reachable by a sequence of ϵ-transitions reading letter σ and one final $\uparrow$-transition reading σ. Formally

$$f_\bot(p, p') = \ \bot$$

$$f_\top(p, p') = \begin{cases} \bot & \text{if } p \in F \text{ or } p' \in F \\ \top & \text{otherwise} \end{cases}$$

$$C_p^\sigma = \left\{ p' \;\middle|\; \begin{array}{l} \exists t_0, t_1, \ldots, t_n \in P^+ \text{ such that} \\ t_0 = p',\ t_n = p, \\ \forall 0 < i < n,\ (\epsilon, t_i) \in \delta(t_{i-1}, \sigma),\ \text{and} \\ (\uparrow, p_n) \in \delta(p_{n-1}, \sigma) \end{array} \right\}$$

Now η is defined for every state in Q as follows (recall that $\mathcal{A}$ is a word automaton, hence we use

directions 0 and 1 in the definition of η, as $\Sigma = \{a\}$, we omit the letter a from the definition of η).

$$\eta(p,d,\alpha) = \bigvee_{p' \in P} \bigvee_{\beta \in \{\bot,\top\}} (0,(p,p',d,\beta)) \wedge (0,(p',d,\beta)) \vee \bigvee_{\upsilon \in \Upsilon} \bigvee_{(\upsilon,p') \in \delta(p,L_\tau(d))} (1,(p',\rho_\tau(d,\upsilon),\bot)) \vee \bigvee_{\langle \epsilon,p' \rangle \in \delta(p,L_\tau(d))} (0,(p',d,\bot))$$

$$\eta(p_1,p_2,d,\alpha) = \bigvee_{\langle \epsilon,p' \rangle \in \delta(p_1,L_\tau(d))} (0,(p',p_2,d,f_\alpha(p',p_2))) \vee \bigvee_{p' \in P} \bigvee_{\beta_1+\beta_2=\alpha} (0,(p_1,p',d,f_{\beta_1}(p_1,p')) \wedge (0,(p',p_2,d,f_{\beta_2}(p',p_2))) \vee \bigvee_{\upsilon \in \Upsilon} \bigvee_{\langle \upsilon,p' \rangle \in \delta(p_1,L_\tau(d))} \bigvee_{p'' \in C_{p_2}^{L_\tau(d)}} (1,(p',p'',\rho_\tau(d,\upsilon),f_\alpha(p',p'')))$$

Finally, we replace every state of the form $\{(p,p,d,\alpha) \mid$ either $p \in P$ and $\alpha = \bot$ or $p \in F$ and $\alpha = \top\}$ by **true**.

Claim 2.4.4 *$L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$*

The proof is very similar to the proof in [PV03]. For the sake of completeness, it is included with the necessary adaptations in Appendix 2.A.

The emptiness of an εABW can be determined in linear space [EL86]. For an εABW $\mathcal{A}$ with one letter alphabet we have the following.

Theorem 2.4.5 [VW86b] *Given an εABW over 1-letter alphabet $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ we can check whether $L(\mathcal{A})$ is empty in time $O(|Q| \cdot |\eta|)$ and space $O(|Q|)$.*

In [PV03] we show that because of the special structure of this εABW, its emptiness can be decided in time $O(|\eta|)$ and space $O(|Q|)$. □

We show now that the emptiness problem for 2NBP is EXPTIME-complete. The upper bound follows immediately from the exponential time algorithm for the emptiness for 2APT [Var98]. For the lower bound we use the EXPTIME-hard problem of whether a linear space alternating Turing machine accepts the empty tape [CKS81]. We reduce this question to the emptiness of a 2NBP with a polynomial number of states.

Theorem 2.4.6 *Consider a 2NBP $\mathcal{P} = \langle \Sigma, P, p_0, \delta, F \rangle$. The emptiness problem of $\mathcal{P}$ is EXPTIME-complete.*

Proof: We use the definitions given in Subsection 2.2.6 of the alternating Turing machine.

We encode the full run tree of M into the labeling of the full infinite binary tree. We construct a 2NBP that reads an input tree and checks that it is indeed a correct encoding of the run tree of M. In a correct encoding of the run tree of M the 2NBP checks that there exists an accepting pruning tree.

We explain now how the labeling of the full binary tree is used to encode the run tree of M. Let $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ be a configuration and $\sharp \cdot \sigma_1^l \cdots \sigma_{f_n}^l$ be its left successor. We set σ_0 and σ_0^l to $\sharp$. Formally, let $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma)$ and let $next_l : V^3 \to V$ where $next_l(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$ denotes our expectation for σ_i^l. We define $next_l(\sigma, \sharp, \sigma') = \sharp$ and

$$next_l(\sigma, \sigma', \sigma'') = \begin{cases} \sigma' & \{\sigma, \sigma', \sigma''\} \subseteq \{\sharp\} \cup \Gamma \\ \sigma' & \sigma'' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', R) \\ (s', \sigma') & \sigma'' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', L) \\ \sigma' & \sigma = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', L) \\ (s', \sigma') & \sigma = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', R) \\ \gamma' & \sigma' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', \alpha) \end{cases}$$

The expectation $next_r : V^3 \to V$ for the letters in the right successor is defined analogously.

The run tree of M is encoded in the full binary tree as follows. Every configuration is encoded by a string of length $f(n)+1$ in $\{\sharp\} \times \Gamma^* \times (S \times \Gamma) \times \Gamma^*$. The encoding of a configuration $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ starts in a node x that is labeled by $\sharp$. The 0 successor of x, namely $0 \cdot x$, is labeled by σ_1 and so on until $0^{f(n)} \cdot x$ that is labeled by $\sigma_{f(n)}$. The configuration $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ has its right successor $\sharp \cdot \sigma_1^r \cdots \sigma_{f(n)}^r$ and its left successor $\sharp \cdot \sigma_1^l \cdots \sigma_{f(n)}^l$. The encoding of $\sharp \cdot \sigma_1^r \cdots \sigma_{f(n)}^r$ starts in $1 \cdot 0^{f(n)} \cdot x$ (that is labeled by $\sharp$) and the encoding of $\sharp \cdot \sigma_1^l \cdots \sigma_{f(n)}^l$ starts in $0 \cdot 0^{f(n)} \cdot x$ (that is labeled by $\sharp$). We also demand that every node be labeled by its direction. This way we can infer from the label of the node labeled by $\sharp$ whether its the first letter in the left successor or the first letter in the right successor. For example, the root of the tree is labeled by $\langle \perp, \sharp \rangle$, the node 0 is labeled by $\langle 0, (s_0, b) \rangle$ and for every $1 < i \leq f(n)$ the node 0^i is labeled by $\langle 0, b \rangle$ (here b stands for the blank symbol). We do not care about the labels of other nodes. Thus, the labeling of 'most' nodes in the tree does not interest us.

The 2NBP reads an infinite binary tree. All trees whose labeling does not conform to the above are rejected. A tree whose labeling is a correct encoding of the run tree of M is accepted only if there exists an accepting pruning tree. Thus, the language of the 2NBP is not empty iff the Turing machine M accepts the empty tape.

In order to check that the input tree is a correct encoding of the run tree of M, the 2NBP has to check that every configuration is followed by its successor configurations. This is done by remembering three letters from the configuration, going $f(n)$ steps forward and checking that the $next_l$ or $next_r$ letter is written in the next configuration. Then the 2NBP returns to the first configuration and updates its three letter memory to include a new letter.

We plunge into the details. The 2NBP has two main modes of operation. In *forward* mode, the 2NBP checks that the next (right or left) configuration is indeed the correct successor. Then it moves to check the next configuration. If it reaches an accepting configuration, this means that the currently scanned pruning tree may still be accepting. Then it moves to *backward* mode and remembers that it should check other universal branches. If it reaches a rejecting configuration, this means that the currently scanned pruning tree is rejecting. The 2NBP has to move to the next pruning tree. It moves to *backward* mode and remembers that it has to check other existential branches. In *backward universal* mode, the 2NBP goes backward until it gets to a universal configuration and the only configuration to be visited below it is the left successor. Then the 2NBP goes back to forward mode but remembers that the next configuration to visit is the right successor. If the root is reached in backward universal mode then there are no more branches to check, the

pruning tree is accepting and the 2NBP accepts. In *backward existential* mode, the 2NBP goes backward until it gets to an existential configuration and the only configuration to be visited below it is the left successor. Then the 2NBP goes to forward mode but remembers that the next configuration to visit is the right successor. If the root is reached in backward existential mode then there are no more pruning trees to check and the 2NBP rejects.

Formally, we have $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$ where

- $\Sigma = \{0, 1, \perp\} \times (\{\sharp\} \cup \Gamma \cup (S \times \Gamma))$.

  Thus, the letters are pairs consisting of a direction and either a $\sharp$, a tape symbol of M, or a tape symbol of M marked by a state of M.

- $P = F \cup B \cup I \cup \{acc\}$ where F is the set of forward states, B is the set of backward states, and I is the set of states that check that the tree starts from the initial configuration of M. All three sets are defined formally below. The state acc is an accepting sink.

- $F = \{acc\}$.

The transition function δ and the initial state p_0 are given below.

We start with forward mode. In forward mode every state is flagged by either l or r, signaling whether the next configuration to be checked is the left successor or the right successor of the current configuration. The 2NBP starts by memorizing the current location it is checking and the environment of this location (that is for checking location i, memorize the letters in locations $i-1$, i, and $i+1$). For checking the left (resp. right) successor it continues $f(n) - i$ steps in direction 0 then it progresses one step in direction 0 (resp. 1) and then takes i steps in direction 0. Finally, it checks that the letter it is reading is indeed the $next_l$ (resp. $next_r$) successor of the memorized environment. It then goes $f(n)-1$ steps back, increases the location that it is currently checking and memorizes the environment of the new location. It continues zigzagging between the two configurations until completing the entire configuration and then it starts checking the next.

Thus, the forward states are $F = \{f\} \times \{l, r\} \times [f(n)] \times V^3 \times [f(n)] \times \{x, v\} \times \{0, 1, \perp\}$. Every state is flagged by f and either r or l (next configuration to be checked is either right or left successor). Then we have the current location $i \in [f(n)]$ we are trying to check, the environment $(\sigma, \sigma', \sigma'') \in V^3$ of this location. Then a counter for advancing $f(n)$ steps. Finally, we have x for *still-checking* and v for *checked* (and going backward to the next letter). We also memorize the direction we went to in order to check that every node is labeled by its direction (thus, we have 0 or 1 for forward moves and $\perp$ for backward moves).

The transition of these states is as follows.

- For $0 \leq i \leq f(n)$ and $0 \leq j < f(n)$ we have

  $$\delta(\langle f, d, i, \sigma, \sigma', \sigma'', j, x, \Delta \rangle, \langle \Delta, \sigma''' \rangle) = \begin{cases} \{(1, \langle f, d, i, \sigma, \sigma', \sigma'', j+1, x, 1 \rangle)\} & i + j = f(n) \text{ and } d = r \\ \{(0, \langle f, d, i, \sigma, \sigma', \sigma'', j+1, x, 0 \rangle)\} & \text{Otherwise} \end{cases}$$

  Continue going forward while increasing the counter. If reached the end of configuration and next configuration is the right configuration go in direction 1. Otherwise go in direction 0.

- For $0 \leq i \leq f(n)$ we have

$$\delta(\langle f,d,i,\sigma,\sigma',\sigma'',f(n),x,\Delta\rangle,\langle\Delta,\sigma'''\rangle) = \begin{cases} \emptyset & \sigma''' \neq next_d(\sigma,\sigma',\sigma'') \\ \{(\uparrow,\langle f,d,(i+1)_{f(n)},\sigma',\sigma'',\bot,f(n)-1,v,\bot\rangle)\} & \sigma''' = next_d(\sigma,\sigma',\sigma'') \end{cases}$$

If σ''' is not the $next_d$ letter, then abort. Otherwise, change the mode to v and start going back. Push σ' and σ'' to the first two memory locations and empty the third memory location.

- For $0 \leq i \leq f(n)$ and $1 < j \leq f(n)$ we have

  $\delta(\langle f,d,i,\sigma,\sigma',\bot,j,v,\bot\rangle,\langle\Delta,\sigma''\rangle) = \{(\uparrow,\langle f,d,i,\sigma,\sigma',\bot,j-1,v,\bot\rangle)\}$.

  Continue going backward while updating the counter.

- For $0 \leq i \leq f(n)$ we have

  $$\delta(\langle f,d,i,\sigma,\sigma',\bot,1,v,\bot\rangle,\langle\Delta,\sigma''\rangle) = \begin{cases} \{(\uparrow,\langle b_\forall,\bot,x\rangle)\} & \sigma'' \in F_a \times \Gamma \\ \{(\uparrow,\langle b_\exists,\bot,x\rangle)\} & \sigma'' \in F_r \times \Gamma \\ \{(\epsilon,\langle f,d,i,\sigma,\sigma',\sigma'',0,x,\bot\rangle)\} & \text{Otherwise} \end{cases}.$$

  Stop going backward. If the configuration that is checked is either accepting or rejecting go to backward mode (recall that the configuration is already verified as the correct successor of the previous configuration). Otherwise memorize the third letter of the environment and initialize the counter to 0.

- $\delta(\langle f,d,0,\sharp,\sharp,\bot,0,x,\Delta\rangle,\langle\Delta,\sigma\rangle) = \{(0,\langle f,d,0,\sharp,\sharp,\sigma,1,x,0\rangle)\}$

  This is the first forward state after backward mode and after the initial phase. It starts checking the first letter of the configuration. The 2NBP already knows that the letter it has to check is $\sharp$, it memorizes the current letter (the third letter of the environment) and moves forward while updating the counter.

  Note that also the first letter is marked as $\sharp$, this is because when checking location 0 of a configuration we are only checking that the length of the configuration is $f(n)+1$ and that after $f(n)+1$ locations there is another $\sharp$.

Backward mode (either universal or existential) is again flagged by l or r, signaling whether the last configuration the 2NBP saw was the left or right successor. Backward mode starts in a node labeled by a state of M. As the 2NBP goes backward, whenever it passes a $\sharp$ it memorizes its direction. When the 2NBP gets again to a letter that is marked with a state of M, if the memorized direction is l and the type of the state the 2NBP is reading matches the type of backward mode (universal state of M and backward universal or existential state of M and backward existential) then the 2NBP continues going up until the $\sharp$, then it moves to forward mode again (marked by r). Otherwise (i.e. if the memorized direction is r or the type of the state the 2NBP is reading does not match the type of backward mode) then the 2NBP stays in backward mode, when it passes the next $\sharp$ it memorizes the current direction, and goes on moving backward. When returning to the root in backward existential mode, this means that the 2NBP is trying to find a new pruning tree. As no such pruning tree exists the 2NBP rejects. When returning to the root in backward universal mode, this means that all universal choices of the currently explored pruning tree were checked and found accepting. Thus, the pruning tree is accepting and the 2NBP accepts.

The set of backward states is $B = \{b_\forall, b_\exists\} \times \{l, r, \bot\} \times \{x, v\}$. Every state is flagged by $\forall$ (for universal) or $\exists$ (for existential) and by either l or r (the last configuration seen is left successor or right successor, or $\bot$ for unknown). Finally, every state is flagged by either x or v. A state marked

by v means that the 2NBP is about to move to forward mode and that it is just going backward until the $\sharp$.

The transition of backward states is as follows.

- $\delta(\langle b_\forall, d, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle b_\forall, l, x\rangle)\} & \sigma = \sharp \text{ and } \Delta = 0 \\ \{(\uparrow, \langle b_\forall, r, x\rangle)\} & \sigma = \sharp \text{ and } \Delta = 1 \\ \{(\epsilon, acc)\} & \Delta = \bot \\ \{(\uparrow, \langle b_\forall, l, v\rangle)\} & \sigma \in S_u \times \Gamma \text{ and } d = l \\ \{(\uparrow, \langle b_\forall, d, x\rangle)\} & \text{Otherwise} \end{cases}$

  In backward universal mode reading a $\sharp$ we memorize its direction. If reading the root, we accept. If reading a universal state of M and the last configuration was the left successor then change the x to v. Otherwise, just keep going backward.

- $\delta(\langle b_\exists, d, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle b_\exists, l, x\rangle)\} & \sigma = \sharp \text{ and } \Delta = 0 \\ \{(\uparrow, \langle b_\exists, r, x\rangle)\} & \sigma = \sharp \text{ and } \Delta = 1 \\ \emptyset & \Delta = \bot \\ \{(\uparrow, \langle b_\exists, l, v\rangle)\} & \sigma \in S_e \times \Gamma \text{ and } d = l \\ \{(\uparrow, \langle b_\forall, d, x\rangle)\} & \text{Otherwise} \end{cases}$

  In backward existential mode reading a $\sharp$ we memorize its direction. If reading the root, we reject. If reading an existential state of M and the last configuration was the left successor then change x to v. Otherwise, just keep going backward.

- $\delta(\langle b, l, v\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle b, l, v\rangle)\} & \sigma \neq \sharp \\ \{(\varepsilon, \langle f, r, 0, \sharp, \sharp, \bot, 0, x, 0\rangle)\} & \sigma = \sharp \end{cases}$

  In backward mode marked by v we go backward until we read $\sharp$. When reading $\sharp$ we return to forward mode. The next configuration to be checked is the right successor. The location we are checking is location 0, thus the letter before is not interesting and is filled by $\sharp$. The counter is initialized to 0.

Finally, the set I of 'initial' states makes sure that the first configuration in the tree is indeed $\sharp \cdot (s_0, b) \cdot b^{f(n)-1}$. When finished checking the first configuration $\mathcal{S}$ returns to the node 0 and moves to forward mode.

Formally, $I = \{i\} \times [f(n)] \times \{x, v\}$ with transition as follows.

- $\delta(\langle i, 0, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(0, \langle i, 1, x\rangle)\} & \sigma = \sharp \text{ and } \Delta = \bot \\ \emptyset & \text{Otherwise} \end{cases}$

  Make sure that the root is labeled by $\langle \bot, \sharp\rangle$.

- $\delta(\langle i, 1, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(0, \langle i, 2, x\rangle)\} & \sigma = (s_0, b) \text{ and } \Delta = 0 \\ \emptyset & \text{Otherwise} \end{cases}$

  Make sure that the first letter is (s_0, b).

- For $1 < j < f(n)$ we have $\delta(\langle i, j, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(0, \langle i, j+1, x\rangle)\} & \sigma = b \text{ and } \Delta = 0 \\ \emptyset & \text{Otherwise} \end{cases}$

  Make sure that all other letters are b.

- $\delta(\langle i, f(n), x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle i, f(n)-1, v\rangle)\} & \sigma = b \text{ and } \Delta = 0 \\ \emptyset & \text{Otherwise} \end{cases}$

  Make sure that the last letter is b. The first configuration is correct, start going back to node 0. Change x to v.

- For $2 < j < f(n)$ we have $\delta(\langle i, j, v\rangle, \langle \Delta, \sigma\rangle) = \{(\uparrow, \langle i, j-1, v\rangle)\}$

  Continue going backward while updating the counter.

- $\delta(\langle i, 2, v\rangle, \langle 0, \sigma\rangle) = \{(\uparrow, \langle f, l, 0, \sharp, \sharp, \bot, 0, x, 0\rangle)\}$.

  Finished checking the first configuration. Go up to node 0 in the first state of forward mode.

Last but not least the initial state is $p_0 = \langle i, 0, x\rangle$.

Finally, we analyze the reduction. Given an alternating Turing machine with n states and alphabet of size m we get a 2NBP with $O(n \cdot m)$ states, that reads an alphabet with $O(n \cdot m)$ letters. The 2NBP is actually deterministic. Clearly, the reduction is polynomial.

We note that instead of checking emptiness of $\mathcal{P}$, we can check the membership of some correct encoding of the run tree of M in the language of $\mathcal{P}$. However, the transducer that generates a correct encoding of M is exponential. □

We note that the membership problem for 2-way alternating Büchi automata on trees is EXPTIME-complete. Indeed, CTL model checking of pushdown systems, proven to be EXPTIME-hard in [Wal00], can be reduced to the membership problem of a regular tree in the language of a 2ABT. Given a pushdown system $R = \langle \Sigma, V, Q, L, T, q_0, x_0\rangle$ and a CTL formula φ, we can construct a graph automaton $\mathcal{S}$ accepting the set of graphs that satisfy φ [KVW00]. This graph automaton is linear in φ and it uses the Büchi acceptance condition. Using the construction Section 2.3, CTL model checking then reduces to the membership problem of $\langle V^*, \tau_V\rangle$ in the language of a 2ABT. EXPTIME-hardness follows. Thus, path automata capture the computational difference between linear and branching specifications.

2.5 Model-Checking Linear-Time Properties

In this section we solve the LTL model-checking problem by a reduction to the membership problem of 2NBP. We start by demonstrating our technique on LTL model checking of pushdown systems. Then we show how to extend it to prefix-recognizable systems. For an LTL formula φ, we construct a 2NBP that navigates through the full infinite V-tree and simulates a computation of the rewrite system that does not satisfy φ. Thus, our 2NBP accepts the V-tree iff the rewrite system does not satisfy the specification. Then, we use the results in Section 2.4: we check whether the given V-tree is in the language of the 2NBP and conclude whether the system satisfies the property. For pushdown systems we show that the tree $\langle V^*, \tau_V\rangle$ gives sufficient information in order to let the 2NBP simulate transitions. For prefix-recognizable systems the label is more complex and reflects the membership of a node x in the regular expressions that are used in the transition rules and the regular labeling.

2.5.1 Pushdown Graphs

Recall that in order to apply a rewrite rule of a pushdown system from configuration (q, x), it is sufficient to know q and the first letter of x. We construct a 2NBP $\mathcal{P}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{P}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{P}$ consults its current state and the label of the node it reads (note that $dir(x)$ is the first letter of x). Formally, we have the following.

Theorem 2.5.1 *Given a pushdown system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$ and an LTL formula φ, there is a 2NBP $\mathcal{P}$ on V-trees such that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff $G_R \not\models \varphi$. The automaton $\mathcal{P}$ has $|Q| \cdot \|T\| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Proof: According to Theorem 2.2.8, there is an NBW $\mathcal{S}_{\neg\varphi} = \langle 2^{AP}, W, \eta_{\neg\varphi}, w_0, F \rangle$ such that $L(\mathcal{S}_{\neg\varphi}) = (2^{AP})^\omega \setminus L(\varphi)$. The 2NBP $\mathcal{P}$ tries to find a trace in G_R that satisfies $\neg\varphi$. The 2NBP $\mathcal{P}$ runs $\mathcal{S}_{\neg\varphi}$ on a guessed (q_0, x_0)-computation in R. Thus, $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff there exists an (q_0, x_0)-trace in G_R accepted by $\mathcal{S}_{\neg\varphi}$. Such a (q_0, x_0)-trace does not satisfy φ, and it exists iff $R \not\models \varphi$. We define $\mathcal{P} = \langle \{V \cup \{\bot\}, P, \delta, p_0, \alpha \rangle$, where

- $P = W \times Q \times \mathit{heads}(T)$, where $\mathit{heads}(T) \subseteq V^*$ is the set of all prefixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{P}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that R with initial configuration $(q, y \cdot x)$ is accepted by $\mathcal{S}^w_{\neg\varphi}$. In particular, when $y = \varepsilon$, then R with initial configuration (q, x) needs to be accepted by $\mathcal{S}^w_{\neg\varphi}$. States of the form $\langle w, q, \varepsilon \rangle$ are called *action states*. From these states $\mathcal{S}$ consults $\eta_{\neg\varphi}$ and T in order to impose new requirements on $\langle V^*, \tau_V \rangle$. States of the form $\langle w, q, y \rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{P}$ only navigates downwards y to reach new action states.
- The transition function δ is defined for every state in $\langle w, q, x \rangle \in S \times Q \times \mathit{heads}(T)$ and letter in $A \in V$ as follows.
  - $\delta(\langle w, q, \epsilon \rangle, A) = \{(\uparrow, \langle w', q', y \rangle) \ : \ w' \in \eta_{\neg\varphi}(w, L(q, A)) \text{ and } \langle q, A, y, q' \rangle \in T\}$.
  - $\delta(\langle w, q, y \cdot B \rangle, A) = \{(B, \langle w, q, y \rangle)\}$.

  Thus, in action states, $\mathcal{P}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to $\eta_{\neg\varphi}$. In navigation states, $\mathcal{P}$ needs to go downwards $y \cdot B$, so it continues in direction B.
- $p_0 = \langle w_0, q_0, x_0 \rangle$. Thus, in its initial state $\mathcal{P}$ checks that R with initial configuration (q_0, x_0) contains a trace that is accepted by $\mathcal{S}$ with initial state w_0.
- $\alpha = \{\langle w, q, \epsilon \rangle \ : \ w \in F \text{ and } \ q \in Q\}$. Note that only action states can be accepting states of $\mathcal{P}$.

We show that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \not\models \varphi$. Assume first that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $(p_0, x_0), (p_1, x_1), \ldots$ of $\mathcal{P}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subsequence of action states $(p_{i_1}, x_{i_1}), (p_{i_2}, x_{i_2}), \ldots$. As the run is accepting and only action states are accepting states, we know that this subsequence is infinite. Let $p_{i_j} = \langle w_{i_j}, q_{i_j}, \varepsilon \rangle$. By the definition of δ,

the sequence $(q_{i_1}, x_{i_1}), (q_{i_2}, x_{i_2}), \ldots$ corresponds to an infinite path in the graph G_R. Also, by the definition of α, the run $w_{i_1}, w_{i_2}, \ldots$ is an accepting run of $\mathcal{S}_{\neg\varphi}$ on the trace of this path. Hence, G_R contains a trace that is accepted by $\mathcal{S}_{\neg\varphi}$, thus $R \not\models \varphi$.

Assume now that $R \not\models \varphi$. Then, there exists a path $(q_0, x_0), (q_1, x_1), \ldots$ in G_R whose trace does not satisfy φ. There exists an accepting run $w_0, w_1, \ldots$ of $\mathcal{S}_{\neg\varphi}$ on this trace. The combination of the two sequences serves as the subsequence of action states in an accepting run of $\mathcal{P}$. It is not hard to extend this subsequence to an accepting run of $\mathcal{P}$ on $\langle V^*, \tau_V \rangle$. □

2.5.2 Prefix-Recognizable Graphs

We now turn to consider prefix-recognizable systems. Again a configuration of a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ consists of a state in Q and a word in V^*. So, the store content is still a node in the tree V^*. However, in order to apply a rewrite rule it is not enough to know the direction of the node. Recall that in order to represent the configuration $(q, x) \in Q \times V^*$, our 2NBP memorizes the state q as part of its state space and it reads the node $x \in V^*$. In order to apply the rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$, the 2NBP has to go up the tree along a word $y \in \alpha_i$. Then, if $x = y \cdot z$, it has to check that $z \in \beta_i$, and finally guess a word $y' \in \gamma_i$ and go downwards y' to $y' \cdot z$. Finding a prefix y of x such that $y \in \alpha_i$, and a new word $y' \in \gamma_i$ is not hard: the 2NBP can emulate the run of the automaton $\mathcal{U}_{\alpha_i}$ while going up the tree and the run of the automaton $\mathcal{U}_{\gamma_i}$ backwards while going down the guessed y'. How can the 2NBP know that $z \in \beta_i$? In Subsection 2.3.2 we allowed the 2APT to branch to two states. The first, checking that $z \in \beta_i$ and the second, guessing y'. With 2NBP this is impossible and we provide a different solution. Instead of labeling each node $x \in V^*$ only by its direction, we can label it also by the regular expressions β for which $x \in \beta$. Thus, when the 2NBP runs $\mathcal{U}_{\alpha_i}$ up the tree, it can tell, in every node it visits, whether z is a member of β_i or not. If $z \in \beta_i$, the 2NBP may guess that time has come to guess a word in γ_i and run $\mathcal{U}_{\gamma_i}$ down the guessed word.

Thus, in the case of prefix-recognizable systems, the nodes of the tree whose membership is checked are labeled by both their directions and information about the regular expressions β. Let $\{\beta_1, \ldots, \beta_n\}$ be the set of regular expressions β_i such that there is a rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \in T$. Let $\mathcal{D}_{\beta_i} = \langle V, D_{\beta_i}, \eta_{\beta_i}, q^0_{\beta_i}, F_{\beta_i} \rangle$ be the deterministic automaton for the reverse of the language of β_i. For a word $x \in V^*$, we denote by $\eta_{\beta_i}(x)$ the unique state that $\mathcal{D}_{\beta_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{\beta_i}$. For a letter $\sigma \in \Sigma$, let $\sigma[i]$, for $i \in \{0, \ldots n\}$, denote the i-th element in σ (that is, $\sigma[0] \in V$ and $\sigma[i] \in D_{\beta_i}$ for $i > 0$). Let $\langle V^*, \tau_\beta \rangle$ denote the Σ-labeled V-tree such that $\tau_\beta(\epsilon) = \langle \bot, q^0_{\beta_1}, \ldots, q^0_{\beta_n} \rangle$, and for every node $A \cdot x \in V^+$, we have $\tau_\beta(A \cdot x) = \langle A, \eta_{\beta_1}(A \cdot x), \ldots, \eta_{\beta_n}(A \cdot x) \rangle$. Thus, every node x is labeled by $dir(x)$ and the vector of states that each of the deterministic automata reach after reading x. Note that $\tau_\beta(x)[i] \in F_{\beta_i}$ iff x is in the language of β_i. Note also that $\langle V^*, \tau_\beta \rangle$ is a regular tree whose size is exponential in the sum of the lengths of the regular expressions $\beta_1, \ldots, \beta_n$.

Theorem 2.5.2 *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and an LTL formula φ, there is a 2NBP $\mathcal{P}$ such that $\mathcal{P}$ accepts $\langle V^*, \tau_\beta \rangle$ iff $R \not\models \varphi$. The automaton $\mathcal{P}$ has $|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot |T| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Proof: As before we use the NBW $\mathcal{S}_{\neg\varphi} = \langle 2^{AP}, W, \eta_{\neg\varphi}, w_0, F \rangle$.

We define $\mathcal{P} = \langle \Sigma, P, \delta, p_0 \alpha \rangle$ as follows.

- $\Sigma = V \times \Pi_{i=1}^{n} D_{\beta_i}$.
- $P = \{\langle w, q, t_i, s\rangle \mid w \in W,\ q \in Q,\ t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \in T,\ \text{ and } s \in Q_{\alpha_i} \cup Q_{\gamma_i}\}$

  Thus, $\mathcal{P}$ holds in its state a state of $\mathcal{S}_{\neg\varphi}$, a state in Q, the current rewrite rule being applied, and the current state in Q_α or Q_γ. A state $\langle w, q, \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, s\rangle$ is an action state if s is the initial state of $\mathcal{U}_{\gamma_i}$, that is $s = q^0_{\gamma_i}$. In action states, $\mathcal{P}$ chooses a new rewrite rule $t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q'\rangle$. Then $\mathcal{P}$ updates the $\mathcal{S}_{\neg\varphi}$ component according to the current location in the tree and moves to $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$. Other states are navigation states. If $s \in Q_{\gamma_i}$ is a state in $\mathcal{U}_{\gamma_i}$ (that is not initial), then $\mathcal{P}$ chooses a direction in the tree, a predecessor of the state in Q_{γ_i} reading the chosen direction, and moves in the chosen direction. If $s \in Q_{\alpha_i}$ is a state of $\mathcal{U}_{\alpha_i}$ then $\mathcal{P}$ moves up the tree (towards the root) while updating the state of $\mathcal{U}_{\alpha_i}$. If $s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ and $\tau(x)[i] \in F_{\beta_i}$ marks the current node x as a member of the language of β_i then $\mathcal{P}$ moves to some accepting state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$ (recall that initial states and accepting states have no incoming / outgoing edges respectively).
- The transition function δ is defined for every state in P and letter in $\Sigma = V \times \Pi_{i=1}^{n} D_{\beta_i}$ as follows.

$$\delta(\langle w, q, t_i, s\rangle, \sigma) = \begin{cases} \left\{ \uparrow, (\langle w, q, t_i, s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \\ s' \in \eta_{\alpha_i}(s, \sigma[0]) \end{array} \right\} \cup \\ \left\{ (\epsilon, \langle w, q, t_i, s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, \\ s \in F_{\alpha_i},\ s' \in F_{\gamma_i}, \\ \text{and } \sigma[i] \in F_{\beta_i} \end{array} \right\} & s \in Q_\alpha \\ \left\{ (B, \langle w, q, t_i, s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \\ s \in \eta_{\gamma_i}(s', B) \text{ and } B \in V \end{array} \right\} \cup \\ \left\{ (\epsilon, \langle w', q'', t_{i'}, s_0\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, \\ t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q''\rangle, \\ w' \in \eta_{\neg\varphi}(w, L(q, \sigma[0])), \\ s = q^0_{\gamma_i} \text{ and } s_0 = q^0_{\alpha_{i'}} \end{array} \right\} & s \in Q_\gamma \end{cases}$$

  Thus, when $s \in Q_\alpha$ the 2NBP $\mathcal{P}$ either chooses a successor s' of s and goes up the tree or in case s is the final state of $\mathcal{U}_{\alpha_i}$ and $\sigma[i] \in F_{\beta_i}$ then $\mathcal{P}$ chooses an accepting state $s' \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$.

  When $s \in Q_\gamma$ the 2NBP $\mathcal{P}$ either guesses a direction B and chooses a predecessor s' of s reading B or in case $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{P}$ updates the state of $\mathcal{S}_{\neg\varphi}$, chooses a new rewrite rule $t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q''\rangle$ and moves to $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$.
- $p_0 = \langle w_0, q_0, t, x_0\rangle$ where t is an arbitrary rewrite rule.

  Thus, $\mathcal{P}$ navigates down the tree to the location x_0. There, it chooses a new rewrite rule and updates the state of $\mathcal{S}_{\neg\varphi}$ and the Q component accordingly.
- $\alpha = \{\langle w, q, t_i, s\rangle \mid w \in F,\ q \in Q,\ t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ \text{and } s = q^0_{\gamma_i}\}$

  Only action states may be accepting. As initial states have no incoming edges, in an accepting run, every navigation stage is finite.

As before we can show that a trace that violates φ and the rewrite rules used to create this trace can be used to produce a run of $\mathcal{P}$ on $\langle V^*, \tau_\beta \rangle$

Similarly, an accepting run of $\mathcal{P}$ on $\langle V^*, \tau_\beta \rangle$ is used to find a trace in G_R that violates φ. $\square$

We can modify the conversion of 2NBP to ϵABW described in Section 2.4 for this particular problem. Instead of keeping in the state of the ϵABW a component of the direction of the node $A \in V \cup \{\bot\}$ we keep the letter from Σ (that is, the tuple $\langle A, q_1, \ldots, q_n \rangle \in V \times \Pi_{i=1}^n D_{\beta_i}$). When we take a move forward in the guessed direction $B \in V$ we update $\langle A, q_1, \ldots, q_n \rangle$ to $\langle B, \eta_{\beta_1}(q_1, B), \ldots, \eta_{\beta_n}(q_n, B) \rangle$. This way, the state space of the resulting ϵABW does not contain $(\Pi_{i=1}^n D_{\beta_i})^2$ but only $\Pi_{i=1}^n D_{\beta_i}$.

Combining Theorems 2.5.1, 2.5.2, and 2.4.3, we get the following.

Theorem 2.5.3 *The model-checking problem for a rewrite system R and an LTL formula φ is solvable*

- *in time $\|T\|^3 \cdot 2^{O(|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|)}$ when R is a pushdown system.*
- *in time $\|T\|^3 \cdot 2^{O(|\varphi|+|Q_\beta|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|+|Q_\beta|)}$ when R is a prefix-recognizable system. The problem is EXPTIME-hard in $|Q_\beta|$ even for a fixed formula.*

For pushdown systems (the first setting), our complexity coincides with the one in [EHRS00]. In Appendix 2.B, we prove the EXPTIME lower bound in the second setting by a reduction from the membership problem of a linear space alternating Turing machine. Thus, our upper bounds are tight.

2.6 Relating Regular Labeling with Prefix-Recognizability

In this section we consider systems with regular labeling. We show first how to extend our approach to handle regular labeling. Both for branching-time and linear-time, the way we adapt our algorithms to handle regular labeling is very similar to the way we handle prefix-recognizability. In the branching-time framework the 2APT guesses a label and sends a copy of the automaton for the regular label to the root to check its guess. In the linear-time framework we include in the labels of the regular tree also data regarding the membership in the languages of the regular labeling. Based on these observations we proceed to show that the two questions are intereducible. We describe a reduction from μ-calculus (resp., LTL) model checking with respect to a prefix-recognizable system with simple labeling function to μ-calculus (resp., LTL) model checking with respect to a pushdown system with regular labeling. We also give reductions in the other direction. We note that we cannot just replace one system by another, but we also have to adjust the μ-calculus (resp., LTL) formula.

2.6.1 Model-Checking Graphs with Regular Labeling

We start by showing how to extend the construction in Subsection 2.3.2 to include also regular labeling. In order to apply a transition of the graph automaton $\mathcal{S}$, from configuration (q, x) our 2APT $\mathcal{A}$ has to guess a label $\sigma \in \Sigma$, apply the transition of $\mathcal{S}$ reading σ, and send an additional copy to the root that checks that the guess is correct and that indeed $x \in R_{\sigma,q}$. The changes to the construction in Subsection 2.3.1 are similar.

Theorem 2.6.1 *Given a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ where L is a regular labeling function and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\bot\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff G_R satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|Q| \cdot (\|T\| + \|L\|) \cdot |V|)$ states, and its index is the index of $\mathcal{S}$ plus 1.*

Proof: We take the automaton constructed for the case of prefix-recognizable systems with simple labeling $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ and modify slightly its state set P and its transition η.

- $P = P_1 \cup P_2 \cup P_3$ where $P_1 = \{\exists, \forall\} \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma)$ and $P_2 = \{\exists, \forall\} \times T \times Q_\beta$ are just like in the previous proof and $P_3 = \bigcup_{\sigma \in \Sigma} \bigcup_{q \in Q} Q_{\sigma,q}$ includes all the states of the automata for the regular expressions appearing in L.
- The definition of $apply_T$ does not change and so does the transition of all navigation states. In the transition of action states, we include a disjunction that guesses the correct labeling. For a state $(d, w, q, t_i, s) \in P_1$ such that $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$ and $s = q^0_{\gamma_i}$ we have

$$\eta((d, w, q, t_i, s), A) = \bigvee_{\sigma \in \Sigma} \left(q^0_{\sigma,q} \wedge apply_T(\delta(w, \sigma), t_i, s) \right)$$

  For a state $s \in Q_{\sigma,q}$ and letter $A \in V \cup \{\bot\}$ we have

$$\eta(s, A) = \begin{cases} \bigvee_{s' \in \rho_{\sigma,q}(s,A)} (\uparrow, s') & A \neq \bot \\ \textbf{true} & A = \bot \text{ and } s \in F_{\sigma,q} \\ \textbf{false} & A = \bot \text{ and } s \notin F_{\sigma,q} \end{cases}$$

□

Theorem 2.6.2 *The model-checking problem for a pushdown or a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ with a regular labeling L and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V| + \|L\| \cdot |V|$ and k is the index of $\mathcal{S}$.*

We show how to extend the construction in Subsection 2.5.2 to include also regular labeling. We add to the label of every node in the tree V^* also the states of the deterministic automata that recognize the reverse of the languages of the regular expressions of the labels. The navigation through the V-tree proceeds as before, and whenever the 2NBP needs to know the label of the current configuration (that is, in action states, when it has to update the state of $\mathcal{S}_{\neg\varphi}$), it consults the labels of the tree.

Formally, let $\{R_1, \ldots, R_n\}$ denote the set of regular expressions R_i such that there exist some state $q \in Q$ and proposition $p \in AP$ with $R_i = R_{p,q}$. Let $\mathcal{D}_{R_i} = \langle V, D_{R_i}, \eta_{R_i}, q^0_{R_i}, F_{R_i} \rangle$ be the deterministic automaton for the reverse of the language of R_i. For a word $x \in V^*$, we denote by $\eta_{R_i}(x)$ the unique state that $\mathcal{D}_{R_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{R_i}$. For a letter $\sigma \in \Sigma$ let $\sigma[i]$, for $i \in \{0, \ldots, n\}$, denote the i-th element of σ. Let $\langle V^*, \tau_L \rangle$ be the Σ-labeled V-tree such that $\tau_L(\epsilon) = \langle \bot, q^0_{R_1}, \ldots, q^0_{R_n} \rangle$ and for every node $A \cdot x \in V^+$ we have $\tau_L(A \cdot x) = \langle A, \eta_{R_1}(A \cdot x), \ldots, \eta_{R_n}(A \cdot x) \rangle$. The 2NBP $\mathcal{P}$ reads $\langle V^*, \tau_L \rangle$. Note that if the state space of $\mathcal{P}$ indicates that the current state of the rewrite system is q and $\mathcal{P}$ reads the node x, then

for every atomic proposition p, we have that $p \in L(q,x)$ iff $\tau_L(x)[i] \in F_{R_i}$, where i is such that $R_i = R_{p,q}$. In action states, $\mathcal{P}$ needs to update the state of $\mathcal{S}_{\neg\varphi}$, which reads the label of the current configuration. Based on its current state and τ_L, the 2NBP $\mathcal{P}$ knows the letter with which $\mathcal{S}_{\neg\varphi}$ proceeds.

If we want to handle a prefix-recognizable system with regular labeling we have to label the nodes of the tree V^* by both the deterministic automata for regular expressions β_i and the deterministic automata for regular expressions $R_{p,q}$. Let $\langle V^*, \tau_{\beta,L}\rangle$ be the composition of $\langle V^*, \tau_\beta\rangle$ with $\langle V^*, \tau_L\rangle$. Notice that $\langle V^*, \tau_L\rangle$ and $\langle V^*, \tau_{\beta,L}\rangle$ are regular, with $\|\tau_L\| = 2^{O(\|L\|)}$ and $\|\tau_{\beta,L}\| = 2^{O(|Q_\beta|+\|L\|)}$.

Theorem 2.6.3 *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0\rangle$ where L is a regular labeling and an LTL formula φ, there is a 2NBP $\mathcal{S}$ such that $\mathcal{S}$ accepts $\langle V^*, \tau_{\beta,L}\rangle$ iff $R \not\models \varphi$. The automaton $\mathcal{S}$ has $|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot \|T\| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Note that Theorem 2.6.3 differs from Theorem 2.5.2 only in the labeled tree whose membership is checked. Combining Theorems 2.6.3 and 2.4.3, we get the following.

Theorem 2.6.4 *The model-checking problem for a prefix-recognizable system R with regular labeling L and an LTL formula φ is solvable in time $\|T\|^3 \cdot 2^{O(|\varphi|+|Q_\beta|+\|L\|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|+|Q_\beta|+\|L\|)}$.*

For pushdown systems with regular labeling an alternative algorithm is given in Theorem 2.2.4. This, together with the lower bound in [EKS01], implies EXPTIME-hardness in terms of $\|L\|$. Thus, our upper bound is tight.

2.6.2 Prefix-Recognizable to Regular Labeling

We give a reduction from μ-calculus and (resp., LTL) model checking of prefix-recognizable systems to μ-calculus (resp., LTL) model checking of pushdown systems with regular labeling. Given a prefix-recognizable system we describe a pushdown system with regular labeling that is used in both reductions. We then explain how to adjust the μ-calculus or LTL formula.

Theorem 2.6.5 *Given a prefix-recognizable system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0\rangle$, a graph automaton $\mathcal{S}$, and an LTL formula φ, there is a pushdown system $R' = \langle 2^{AP'}, V, Q', L', T', q_0', x_0\rangle$ with a regular labeling function, a graph automaton $\mathcal{S}'$, and an LTL formula φ', such that $R \models \mathcal{S}$ iff $R' \models \mathcal{S}'$ and $R \models \varphi$ iff $R' \models \varphi'$. Furthermore, $|Q'| = |Q| \times |T| \times (|Q_\alpha| + |Q_\gamma|)$, $\|T'\| = O(\|T\|)$, $\|L\| = |Q_\beta|$, $|\mathcal{S}'| = O(|\mathcal{S}|)$, the index of $\mathcal{S}'$ equals the index of $\mathcal{S}$ plus one, and $|\varphi'| = O(|\varphi|)$. The reduction is computable in logarithmic space.*

The idea is to add to the configurations of R labels that would enable the pushdown system to simulate transitions of the prefix-recognizable system. Recall that in order to apply the rewrite rule $\langle q, \alpha, \beta, \gamma, q'\rangle$ from configuration (q,x), the prefix-recognizable system has to find a partition $y \cdot z$ of x such that the prefix y is a word in α and the suffix z is a word in β. It then replaces y by a word $y' \in \gamma$. The pushdown system can remove the prefix y letter by letter, guess whether the remaining suffix z is a word in β, and add y' letter by letter. In order to check the validity of guesses, the system marks every configuration where it guesses that the remaining suffix is a word in β. It then

consults the regular labeling function in order to single out traces in which a wrong guess is made. For that, we add a new proposition, *not_wrong*, which holds in a configuration iff it is not the case that pushdown system guesses that the suffix z is in the language of some regular expression r and the guess turns out to be incorrect. The pushdown system also marks the configurations where it finishes handling some rewrite rule. For that, we add a new proposition, *ch-rule*, which is true only when the system finishes handling some rewrite rule and starts handling another.

The pushdown system R' has four modes of operation when it simulates a transition that follows a rewrite rule $\langle q, \alpha, \beta, \gamma, q' \rangle$. In *delete* mode, R' deletes letters from the store x while emulating a run of $\mathcal{U}_{\alpha_i}$. Delete mode starts from the initial state of $\mathcal{U}_{\alpha_i}$, from which R' proceeds until it reaches a final state of $\mathcal{U}_{\alpha_i}$. Once the final state of $\mathcal{U}_{\alpha_i}$ is reached, R' transitions to *change-direction* mode, where it does not change the store and just moves to a final state of $\mathcal{U}_{\gamma_i}$, and transitions to *write* mode. In write mode, R' guesses letters in V and emulates the run of $\mathcal{U}_{\gamma_i}$ on them backward, while adding them to the store. From the initial state of $\mathcal{U}_{\gamma_i}$ the pushdown system R' transitions to *change-rule* mode, where it chooses a new rewrite rule $\langle q', \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q'' \rangle$ and transitions to delete mode. Note that if delete mode starts in configuration (q, x) it cannot last indefinitely. Indeed, the pushdown system can remove only finitely many letters from the store. On the other hand, since the store is unbounded, write mode can last forever. Hence, traces along which *ch-rule* occurs only finitely often should be singled out.

Singling out of traces is done by the automaton $\mathcal{S}'$ and the formula φ' which restrict attention to traces in which *not_wrong* is always asserted and *ch-rule* is asserted infinitely often.

Formally, R' has the following components

- $AP' = AP \cup \{not\_wrong, ch\text{-}rule\}$.
- $Q' = Q \times T \times (\{ch\text{-}dir, ch\text{-}rule\} \cup Q_\alpha \cup Q_\gamma)$. A state $\langle q, t, s \rangle \in Q'$ maintains the state $q \in Q$ and the rewrite rule t currently being applied. the third element s indicates the mode of R'. Change-direction and change-rule modes are indicated by a marker. In delete and write modes, R' also maintains the current state of $\mathcal{U}_\alpha$ and $\mathcal{U}_\gamma$.
- For every proposition $p \in AP$, we have $p \in L'(q, x)$ iff $p \in L(q, x)$. We now describe the regular expression for the propositions *ch-rule* and *not_wrong*. The proposition *ch-rule* holds in all the configuration in which the system is in change-rule mode. Thus, for every $q \in Q$ and $t \in T$, we have $R_{\langle q,t,ch\text{-}rule\rangle, ch\text{-}rule} = V^*$ and $R_{\langle q,t,\zeta\rangle, ch\text{-}rule} = \emptyset$ for $\zeta \neq ch\text{-}rule$. The proposition *not_wrong* holds in configurations in which we are not in change-direction mode, or configuration in which we are in change-direction mode and the store is in β, thus changing direction is possible in the configuration. Formally, for every $q \in Q$ and $t = \langle q', \alpha, \beta, \gamma, q \rangle \in T$, we have $R_{\langle q,t,ch\text{-}dir\rangle, not\_wrong} = \beta$ and $R_{\langle q,t,\zeta\rangle, not\_wrong} = V^*$ for $\zeta \neq ch\text{-}dir$.
- $q'_0 = \langle q_0, t, ch\text{-}rule \rangle$ for some arbitrary rewrite rule t.

The transition function of R' includes four types of transitions according to the four operation modes. In change-direction mode, in configuration $(\langle q, t, ch\text{-}dir \rangle, x)$ that applies the rewrite rule $t = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$, the system R' does not change x, and moves to a final state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$. In change rule mode, in configuration $(\langle q, t, ch\text{-}rule \rangle, x)$, the system R' does not change x, it chooses a new rewrite rule $t' = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle$, changes the Q component to q', and moves to the initial state $q^0_{\alpha_{i'}}$ of $\mathcal{U}_{\alpha_{i'}}$. In delete mode, in configuration $(\langle q, t, s \rangle, x)$, for $t = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$ and $s \in Q_{\alpha_i}$, the system R' proceeds by either removing one letter from x and continuing the run of $\mathcal{U}_{\alpha_i}$, or if

$s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ then R' leaves x unchanged, and changes s to *ch-dir*. In write mode, in configuration $(\langle q, t, s\rangle, x)$, for $t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle$ and $s \in Q_{\gamma_i}$, the system R' proceeds by either extending x with a guessed symbol from V and continuing the run of $\mathcal{U}_{\gamma_i}$ backward using the guessed symbol, or if $s = q^0_{\gamma_i}$, then R' leaves x unchanged and just replaces s by *ch-rule*. Formally, $T' = T'_{ch\text{-}rule} \cup T'_{ch\text{-}dir} \cup T'_\alpha \cup T'_\gamma$, where

- $T'_{ch\text{-}rule} = \left\{ (\langle q, t, ch\text{-}rule\rangle, A, A, \langle q', t', s\rangle) \;\middle|\; t' = \langle q, \alpha_i, \beta_i, \gamma_i, q'\rangle,\ s = q^0_{\alpha_i} \text{ and } A \in V \right\}$.
- $T'_{ch\text{-}dir} = \{ (\langle q, t, ch\text{-}dir\rangle, A, A, \langle q, t, s\rangle) \mid t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ s \in F_{\gamma_i},\ \text{and } A \in V \}$.

Note that the same letter A is removed from the store and added again. Thus, the store content of the configuration does not change.

- $T'_\alpha = \begin{array}{l} \{ (\langle q, t, s\rangle, A, \epsilon, \langle q, t, s'\rangle) \mid t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ s \in Q_\alpha,\ s' \in \rho_{\alpha_i}(s, A),\ \text{and } A \in V \} \cup \\ \{ (\langle q, t, s\rangle, A, A, \langle q, t, ch\text{-}dir\rangle) \mid t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ s \in Q_\alpha,\ s \in F_{\alpha_i},\ \text{and } A \in V \}. \end{array}$
- $T'_\gamma =$
  $\{ (\langle q, t, s\rangle, A, AB, \langle q, t, s'\rangle) \mid t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ s \in Q_\gamma,\ s \in \rho_{\gamma_i}(s', B),\ \text{and } A, B \in V \} \cup$
  $\left\{ (\langle q, t, s\rangle, A, A, \langle q, t, ch\text{-}rule\rangle) \mid t = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ s \in Q_\gamma,\ s = q^0_{\gamma_i} \text{ and } A \in V \right\}$.

As final states have no outgoing edges, after a state $\langle q, \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, s\rangle$ for $s \in F_{\alpha_i}$ we always visit the state $\langle q, t, ch\text{-}dir\rangle$. Similarly, as initial states have no incoming edges, we always visit the state $\langle q, t, ch\text{-}rule\rangle$ after visiting a state $\langle q, \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, q^0_{\gamma_i}\rangle$.

The automaton $\mathcal{S}'$ adjusts $\mathcal{S}$ to the fact that every transition in R corresponds to multiple transitions in R'. Accordingly, when $\mathcal{S}$ branches universally, infinite navigation stages and states not marked by *not_wrong* are allowed. Dually, when $\mathcal{S}$ branches existentially, infinite navigation stages and states not marked by *not_wrong* are not allowed.

Formally, let $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F\rangle$. We define, $\mathcal{S}' = \langle \Sigma, W', \delta', w_0, \alpha\rangle$ where

- $W' = W \cup (\{\forall, \exists\} \times W)$ Intuitively, when $\mathcal{S}'$ reads configuration (q, x) and transitions to $\exists w$ it is searching for a successor of (q, x) that is accepted by S^w. The state $\exists w$ navigates to some configuration reachable from (q, x) of R' marked by *ch-rule*. Dually, when $\mathcal{S}'$ reads configuration (q, x) and transitions to $\forall w$ it is searching for all successors of (q, x) and tries to ensure that they are accepted by S^w. The state $\forall w$ navigates to all configurations reachable from (q, x) of R' marked by *ch-rule*.
- For every state $w \in W$ and letter $\sigma \in \Sigma$, the transition function δ' is obtained from δ by replacing every atom of the form $\Box w$ by $\Box(\forall w)$ and every atom of the form $\Diamond w$ by $\Diamond(\exists w)$.

  For every state $w \in W$ and letter $\sigma \in \Sigma$, we have

$$\delta'(\forall w, \sigma) = \begin{cases} \mathbf{true} & \sigma \not\models \mathit{not\_wrong} \\ (\varepsilon, w) & \sigma \models \mathit{not\_wrong} \wedge \mathit{ch\text{-}rule} \\ (\Box, \forall w) & \sigma \models \mathit{not\_wrong} \wedge \neg \mathit{ch\text{-}rule} \end{cases}$$

$$\delta'(\exists w, \sigma) = \begin{cases} \mathbf{false} & \sigma \not\models \mathit{not\_wrong} \\ (\varepsilon, w) & \sigma \models \mathit{not\_wrong} \wedge \mathit{ch\text{-}rule} \\ (\Diamond, \exists w) & \sigma \models \mathit{not\_wrong} \wedge \neg \mathit{ch\text{-}rule} \end{cases}$$

- α is obtained from F by including all states in $\{\forall\} \times W$ as the maximal even set and all states in $\{\exists\} \times W$ as the maximal odd set.

Claim 2.6.6 *$G_R \models \mathcal{S}$ iff $G_{R'} \models \mathcal{S}'$*

Proof: Assume that $G_{R'} \models \mathcal{S}'$. Let $\langle T', r' \rangle$ be an accepting run of $\mathcal{S}'$ on $G_{R'}$. We construct an accepting run $\langle T, r \rangle$ of $\mathcal{S}$ on G_R based on the subtree of nodes in T' labeled by states in W (it follows that these nodes are labeled by configurations with state *ch-rule*). Formally, we have the following. We have $r'(\varepsilon) = ((q_0, x_0), w_0)$. We add to T the node ε and label it $r(\varepsilon) = ((q_0, x_0), w_0)$. Given a node $z \in T$ labeled by $r(z) = ((q, x), w)$, it follows that there exists a node $z' \in T'$ labeled by $r'(z') = ((q, x), w)$. Let $\{((q_i, x_i), w_i)\}_{i \in I}$ be the labels of the minimal nodes in T' labeled by states in W. We add $|I|$ successors $\{a_i z\}_{i \in I}$ to z in T and label them $r(a_i z) = ((q_i, x_i), w_i)$. From the definition of R' it follows that $\langle T, r \rangle$ is a valid run of $\mathcal{S}$ on G_R. As every infinite path in T corresponds to an infinite path in T' all whose nodes are marked by configurations marked by *not_wrong* and infinitely many configurations are marked by *ch-rule* it follows that $\langle T, r \rangle$ is an accepting run.

In the other direction, we extend an accepting run tree $\langle T, r \rangle$ of $\mathcal{S}$ on G_R into an accepting run tree of $\mathcal{S}'$ on $G_{R'}$ by adding transitions to $\{\forall, \exists\} \times W$ type states. □

Corollary 2.6.7 *Given a prefix-recognizable system R and a graph automaton $\mathcal{S}$ with n states and index k, we can model check $\mathcal{S}$ with respect to R in time exponential in $n \cdot k \cdot \|T\|$.*

Finally, we proceed to the case of an LTL formula φ. The formula φ' is the implication $\varphi'_1 \rightarrow \varphi'_2$ of two formulas. The formula φ'_1 holds in computations of R' that correspond to real computations of R. Thus, $\varphi'_1 = \Box \mathit{not\_wrong} \wedge \Box\Diamond \mathit{ch\text{-}rule}$. Then, φ'_2 adjusts φ to the fact that a single transition in R corresponds to multiple transitions in R'. Formally, $\varphi'_2 = f(\varphi)$, for the function f defined below.

- $f(p) = p$ for a proposition $p \in AP$
- $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
- $f(a \mathcal{U} b) = (\mathit{ch\text{-}rule} \rightarrow f(a)) \mathcal{U} (\mathit{ch\text{-}rule} \wedge f(b))$
- $f(\bigcirc a) = \bigcirc((\neg \mathit{ch\text{-}rule}) \mathcal{U} (\mathit{ch\text{-}rule} \wedge f(a))$

Claim 2.6.8 *$G_R \models \varphi$ iff $G_{R'} \models \varphi'$*

We first need some definitions and notations. We define a partial function g from traces in $G_{R'}$ to traces in G_R. Given a trace π' in $G_{R'}$, if $\pi' \not\models \varphi'_1$ then $g(\pi')$ is undefined. Otherwise, denote $\pi' = (p'_0, w_0), (p'_1, w_1), \ldots$ and

$$g(\pi') = \begin{cases} (p, w_0), g(\pi'_{\geq 1}) & p'_0 = \langle p, t, \mathit{ch\text{-}rule} \rangle \\ g(\pi'_{\geq 1}) & p'_0 = \langle p, t, \alpha \rangle \text{ and } \alpha \neq \mathit{ch\text{-}rule} \end{cases}$$

Thus, g picks from π' only the configurations marked by *ch-rule*, it then takes the state from Q that marks those configurations and the store. Furthermore given two traces π' and $g(\pi')$ we define a matching between locations in π' in which the configuration is marked by *ch-rule* and the locations in $g(\pi')$. Given a location i in $g(\pi')$ we denote by $ch(i)$ the location in π' of the i-th occurrence of *ch-rule* along π'.

Lemma 2.6.9 *1. For every trace π' of $G_{R'}$, $g(\pi')$ is either not defined or a valid trace of G_R.*

2. The function g is a bijection between domain(g) and the traces of G_R.

3. For every trace π' of $G_{R'}$ such that $g(\pi')$ is defined, we have $(\pi', ch(i)) \models f(\varphi)$ iff $(g(\pi'), i) \models \varphi$

Proof: 1. Suppose $g(\pi')$ is defined, we have to show that it is a trace of G_R. The first pair in π' is $(\langle q_0, t, \textit{ch-rule}\rangle, x_0)$. Hence $g(\pi')$ starts from (q_0, x_0). Assume by induction that the prefix of $g(\pi')$ up to location i is the prefix of some computation in G_R. We show that also the prefix up to location i+1 is a prefix of a computation. Let $(\langle q, t, \textit{ch-rule}\rangle, x)$ be the i-th *ch-rule* appearing in π', then the i-th location in $g(\pi')$ is (q, x). The computation of R' chooses some rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q'\rangle \in T$ and moves to state $\langle q', t_i, s\rangle$ where $s = q^0_{\alpha_i}$. It must be the case that a state $\langle q', t_i, \textit{ch-dir}\rangle$ appears in the computation of R' after location $ch(i)$. Otherwise, the computation is finite and does not interest us. The system R' can move to a state marked by *ch-dir* only from $s \in F_{\alpha_i}$, an accepting state of $\mathcal{U}_{\alpha_i}$. Hence, we conclude that $x = y \cdot z$ where $y \in \alpha_i$. As *not_wrong* is asserted everywhere along π' we know that $z \in \beta_i$. Now R' adds a word y' in γ_i to z and reaches state $(\langle q', t', \textit{ch-rule}\rangle, y' \cdot z)$. Thus, the transition t is possible also in R and can lead from $(q, y \cdot z)$ to $(q', y' \cdot z)$.

2. It is quite clear that g is an injection. As above, given a trace π in G_R we can construct the trace π' in $G_{R'}$ such that $g(\pi') = \pi$.

3. We prove that $(\pi, i) \models \varphi$ iff $(\pi', ch(i)) \models \varphi$ by induction on the structure of φ.

   - For a boolean combination of formulas the proof is immediate.
   - For a proposition $p \in AP$, it follows from the proof above that if in location i in $g(\pi')$ appears state (q, x) then in location $ch(i)$ in π' appears state $(\langle q, t, \textit{ch-rule}\rangle, x)$. By definition $p \in L(q, x)$ iff $p \in L'(\langle q, t, \textit{ch-rule}\rangle, x)$.
   - For a formula $\varphi = \psi_1 \mathcal{U} \psi_2$. Suppose $(g(\pi'), i) \models \varphi$. Then there exists some $j \geq i$ such that $(g(\pi'), j) \models \psi_2$ and for all $i \leq k < j$ we have $(g(\pi'), k) \models \psi_1$. By the induction assumption we have that $(\pi', ch(j)) \models f(\psi_2)$ (and clearly, $(\pi', ch(j)) \models \textit{ch-rule}$), and for all $i \leq j < k$ we have $(\pi', ch(k)) \models \psi_1$. Furthermore, as every location marked by *ch-rule* is associated by the function ch to some location in $g(\pi')$ all other locations are marked by $\neg$*ch-rule*. Hence, $(\pi', ch(i)) \models (\textit{ch-rule} \rightarrow f(\psi_1))\mathcal{U}(f(\psi_2) \wedge \textit{ch-rule})$.

     The other direction is similar.
   - For a formula $\varphi = \bigcirc\psi$ the argument resembles the one above for $\mathcal{U}$.

□

We note that for every trace π' and $g(\pi')$ we have that $ch(0) = 0$. Claim 2.6.8 follows immediately.

If we use this construction in conjunction with Theorem 2.2.4, we get an algorithm whose complexity coincides with the one in Theorem 2.5.3.

Corollary 2.6.10 *Given a prefix-recognizable system R and an LTL formula φ we can model check φ with respect to R in time $O(\|T\|^3) \cdot 2^{O(|Q_\beta|)} \cdot 2^{O(|\varphi|)}$ and space $O(\|T\|^2) \cdot 2^{O(|Q_\beta|)} \cdot 2^{O(|\varphi|)}$.*

Notice, that for LTL, we change the formula itself while for μ-calculus we change the graph automaton resulting from the formula. Consider the following function from μ-calculus formulas to μ-calculus formulas.

- For $p \in AP$ we have $f(p) = \mathit{ch\text{-}rule} \wedge p$.
- $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
- $f(\Box a) = \Box \nu X (f(a) \wedge \mathit{ch\text{-}rule} \vee \neg \mathit{not\_wrong} \vee \neg \mathit{ch\text{-}rule} \wedge \Box X)$
- $f(\Diamond a) = \Diamond \mu X (f(a) \wedge \mathit{ch\text{-}rule} \wedge \mathit{not\_wrong} \vee \neg \mathit{ch\text{-}rule} \wedge \mathit{not\_wrong} \wedge \Diamond X)$
- $f(\mu X a(X)) = \mu X (\mathit{ch\text{-}rule} \wedge f(a(X)))$
- $f(\nu X a(X)) = \nu X (\mathit{ch\text{-}rule} \wedge f(a(X)))$

We claim that $R \models \psi$ iff $R' \models f(\psi)$. However, the alternation depth of $f(\psi)$ my be much larger than that of ψ. For example, $\varphi = \mu X (p \wedge \Box(\neg p \wedge \Box(X \wedge \mu Y (q \vee \Box Y))))$ is alternation free, while $f(\varphi)$ is of alternation depth 3. This kind of transformation is more appropriate with the equational form of μ-calculus where we can declare all the newly added fixpoints as minimal and incur only an increase of 1 in the alternation depth.

We note that since we end up with a pushdown system with regular labeling, it is easy to extend the reduction to start with a prefix-recognizable system with regular labeling. It is left to show the reduction in the other direction.

2.6.3 Regular Labeling to Prefix-recognizable

We show that we can also reduce the problem of μ-calculus (resp., LTL) model checking of pushdown graphs with regular labeling, to the problem of μ-calculus (resp., LTL) model checking of prefix-recognizable graphs. This time we use different modifications of the system. With branching-time formalisms we can mark a letter true by adding a special successor. This is impossible in linear-time.

Theorem 2.6.11 *Given a pushdown system $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ with a regular labeling function and a graph automaton $\mathcal{S}$, there is a prefix-recognizable system $R' = \langle \Sigma, V, Q', T', L', q'_0, x_0 \rangle$ with simple labeling and a graph automaton $\mathcal{S}'$ such that $R \models \mathcal{S}$ iff $R' \models \mathcal{S}'$. Furthermore, $|Q'| = |Q| + |\Sigma|$, $|Q'_\alpha| + |Q'_\gamma| = O(\|T\|)$, and $|Q'_\beta| = \|L\|$. The reduction is computable in logarithmic space.*

Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$. The idea behind the reduction is as follows. The state space of R' is $Q \cup \Sigma$. Given some configuration (q, x) of R' such that (q, x) is labeled by σ in R, we add in R' a transition to configuration (σ, x). Basically, R' is identical to R with these additional transitions.

Given a pushdown system $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ with a regular labeling function, we construct a prefix-recognizable system $R' = \langle \Sigma', V, Q', T', L', q_0, x_0 \rangle$ with simple labeling as follows.

- $\Sigma' = \Sigma \cup \{\sigma_0\}$. The letter σ_0 marks all original configurations of R.
- $Q' = Q \cup \Sigma$. The state σ is labeled by σ.

- $T' = T \cup \{\langle q, \varepsilon, R_{\sigma,q}, \varepsilon, \sigma\rangle \mid \sigma \in \Sigma\}$. We abuse notation and write a transition $\langle q, A, x, q'\rangle \in T$ as a prefix-recognizable transition. In addition, in state q, when the store content is in $R_{\sigma,q}$ we add a transition to state σ.
- For $q \in Q$ and $A \in V$ we set $L'(q, A) = \sigma_0$ and $L'(\sigma, A) = \sigma$.

The graph automaton $\mathcal{S}'$ uses the states of $\mathcal{S}$ in addition to Σ (as states). The transition of a state $\sigma \in \Sigma$ is defined only for the letter σ. For other states, we guess a label σ, use the transition using this letter and check that the current state has a successor labeled σ.

Let $\mathcal{S} = \langle \Sigma, S, \delta, s_0, F\rangle$. We set $\mathcal{S}' = \langle \Sigma', S', \delta', s_0, F\rangle$ where $S' = S \cup \Sigma$ and the transition δ' is defined as follows. For $s \in S$ and $\sigma \in \Sigma$, let $\delta''(s, \sigma)$ denote the formula in $\mathcal{B}^+(\Delta \times S')$ that is obtained from $\delta(s, \sigma)$ by replacing an atom $\Diamond t$ by $\Diamond(t \wedge \sigma_0)$, and an atom $\Box t$ by $\Box(\sigma_0 \rightarrow t)$. Now for every state $s \in S'$ and letter $\alpha \in \Sigma'$ the transition function δ' is defined as follows.

$$\delta'(s, \alpha) = \begin{cases} \textbf{false} & s \in S \text{ and } \alpha \neq \sigma_0 \\ \bigvee_{\sigma \in \Sigma} \Diamond\sigma \wedge \delta''(s, \sigma) & s \in S \text{ and } \alpha = \sigma_0 \\ \textbf{false} & s \in \Sigma \text{ and } \alpha \neq s \\ \textbf{true} & s \in \Sigma \text{ and } \alpha = s \end{cases}$$

Claim 2.6.12 *$G_R \models \mathcal{S}$ iff $G_{R'} \models \mathcal{S}'$*

Proof: We translate an accepting run tree of $\mathcal{S}$ on G_R to an accepting run tree of $\mathcal{S}'$ on $G_{R'}$ by adding visits to the configurations with states in Σ.

In the other direction, we simply remove from the run tree of $\mathcal{S}'$ all states in Σ. □

Theorem 2.6.13 *Given a pushdown system $R = \langle 2^{AP}, V, Q, T, L, q_0, x_0\rangle$ with a regular labeling function and an LTL formula φ, there is a prefix-recognizable system $R' = \langle 2^{AP'}, V, Q', T', L', q'_0, x_0\rangle$ with simple labeling and an LTL formula φ' such that $R \models \varphi$ iff $R' \models \varphi'$. Furthermore, $|Q'| = O(|Q| \cdot |AP|)$, $|Q'_\alpha| + |Q'_\gamma| = O(\|T\|)$, and $|Q'_\beta| = 2^{\|L\|}$ yet the automata for Q'_β are deterministic. The reduction is computable in polynomial space.*

Let $AP = \{p_1, \ldots, p_n\}$ be the set of atomic propositions. The idea behind the reduction is as follows. The state space of R' is $Q \times (\{start\} \cup AP) \times \{\bot, \top\}$. We replace a configuration (q, x) in R by a sequence $\langle(\langle q, start, \bot\rangle, x), (\langle q, p_1, \lambda_1\rangle, x), \ldots, (\langle q, p_n, \lambda_n\rangle, x)\rangle$ of $n+1$ configurations in R', where $\lambda_i = \bot$ if $x \notin R_{q,p_i}$ and $\lambda = \top$ if $x \in R_{q,p_i}$. Each of the last n states corresponds to one of the propositions in AP. The new rewrite rule checks that the marking of $\bot$ and $\top$ is indeed correct by matching the regular expression β of the transition with the regular expression of the proposition. For that, we use two types of transition rules. First, the transition rule $\langle\langle q, p_{i-1}, \zeta\rangle, \epsilon, R_{q,p_i}, \epsilon, \langle q, p_i, \top\rangle\rangle$ marks p_i as true and makes sure that $x \in R_{q,p_i}$. Second, $\langle\langle q, p_{i-1}, \zeta\rangle, \epsilon, \widetilde{R}_{q,p_i}, \epsilon, \langle q, p_i, \bot\rangle\rangle$, where $\widetilde{R}$ is the regular expression for the complement of R, marks p_i as false and makes sure that $x \notin R_{q,p_i}$. The automaton $\widetilde{\mathcal{U}}_{q,p_i}$ that recognizes the language of $\widetilde{R}_{q,p_i}$ may be exponentially larger than $\mathcal{U}_{q,p_i}$. Thus, the system R' may be exponentially larger than R. However, reasoning about the correctness of R' requires the automata for the regular expressions to be deterministic, thus although R' may be exponentially larger than R, the model-checking problem of R' is only exponential in $\|L\|$ and not doubly exponential.

Given a pushdown system $R = \langle 2^{AP}, V, Q, T, L, q_0, x_0\rangle$ with a regular labeling function, we construct a prefix-recognizable system $R' = \langle 2^{AP'}, V, Q', T', L', q'_0, x_0\rangle$ with simple labeling as follows.

- $AP' = AP \cup \{start\}$. The proposition *start* marks the beginning of the sequence of length $n+1$ states of $G_{R'}$ that relates to one state of G_R.
- $Q' = (Q \times AP \times \{\bot, \top\}) \cup (Q \times \{start\})$. A state (q, x) in G_R relates to the sequence that starts with $(\langle q, start\rangle, x)$, continues to $(\langle q, p_1, \lambda_1\rangle, x)$ where λ_1 is as before, and then proceeds to $(\langle q, p_2, \lambda_2\rangle, x)$ and forward until $(\langle q, p_n, \lambda_n\rangle, x)$.
- $L'((q, start), x) = \{start\}$.
  $L'((q, p, \top), x) = \{p\}$.
  $L'((q, p, \bot), x) = \emptyset$.
- $T'_0 = \left\{ \begin{array}{l} \langle\langle q, start\rangle, \epsilon, R_{q,p_1}, \epsilon, \langle q, p_1, \top\rangle\rangle, \\ \langle\langle q, start\rangle, \epsilon, \widetilde{R}_{q,p_1}, \epsilon, \langle q, p_1, \bot\rangle\rangle \end{array} \;\middle|\; q \in Q \right\}$
  For $1 \leq i < n$ we have
  $T'_i = \left\{ \begin{array}{l} \langle\langle q, p_i, \alpha\rangle, \epsilon, R_{q,p_{i+1}}, \epsilon, \langle q, p_{i+1}, \top\rangle\rangle, \\ \langle\langle q, p_i, \alpha\rangle, \epsilon, \widetilde{R}_{q,p_{i+1}}, \epsilon, \langle q, p_{i+1}, \bot\rangle\rangle \end{array} \;\middle|\; \begin{array}{l} q \in Q \text{ and} \\ \alpha \in \{\bot, \top\} \end{array} \right\}$
  $T'_n = \{(\langle q, p_n, \alpha\rangle, A, V^*, \{x\}, \langle q', start\rangle) \mid q \in Q,\ \alpha \in \{\bot, \top\},\ \text{ and } \langle q, A, x, q'\rangle \in T\}$
  Finally, we have $T' = \bigcup_{i=0}^{n} T'_i$. Thus, from a configuration marked by p_i we move to a state marked by p_{i+1} without changing the store contents. We mark p_{i+1} as true or false according to the store. From a configuration marked by p_n we apply a new rewrite rule according to the first letter in the store.
- $q'_0 = (q_0, start)$.

In order to define the LTL formula φ' we define the function f from LTL formulas to LTL formulas as follows. Intuitively, $f(\varphi)$ adjusts φ to the representation of a single state and its labeling by a chain of $|AP|+1$ states (we assume that $|AP| = n$).

- For the proposition $p_i \in AP$ we have $f(p_i) = \bigcirc^i p_i$
- $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
- $f(a \mathcal{U} b) = (start \rightarrow f(a)) \mathcal{U} (f(b) \wedge start)$
- $f(\bigcirc a) = \bigcirc^{n+1} f(a)$

Claim 2.6.14 *$G_R \models \varphi$ iff $G_{R'} \models f(\varphi)$*

Proof: Consider a trace π in G_R that does not satisfy φ. Consider the trace π' where every configuration (q, x) in π is replaced by the sequence $(q, start), (q, p_1, \alpha_1), \ldots, (q, p_n, \alpha_n)$ where α_i is $\top$ iff $x \in R_{p,q}$. It is simple to see that π' is a trace of $G_{R'}$ and that $\pi, i \models \varphi$ iff $\pi', (n+1)i \models f(\varphi)$.

In the other direction, we replace every sequence of $n+1$ configurations by a single configuration. □

If we use this construction in conjunction with Theorem 2.5.3, we get an algorithm whose complexity coincides with the one in [EKS01].

2.7 Realizability and Synthesis

In this section we show that the automata theoretic approach can be used also to solve the realizability and synthesis problems for branching time and linear time specifications of pushdown and prefix-recognizable systems. We start with a definition of the realizability and synthesis problems and then proceed to give algorithms that solve these problems for μ-calculus and LTL.

Given a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a partition of T to $\{T_1, \ldots, T_m\}$, a *strategy* of R is a function $f : Q \times V^* \to [m]$. The function f restricts the graph G_R so that from a configuration $(q, x) \in Q \times V^*$, only $f(q, x)$ transitions are taken. Formally, R and f together define the graph $G_{R,f} = \langle \Sigma, Q \times V^*, \rho, (q_0, x_0), L \rangle$, where $\rho((q, x), (q', y))$ iff $f(q, x)=i$ and there exists $t \in T_i$ such that $\rho_t((q, x), (q', y))$. Given R and a specification ψ (either a graph automaton or an LTL formula), we say that a strategy f of R is *winning* for ψ iff $G_{R,f}$ satisfies ψ. Given R and ψ the problem of *realizability* is to determine whether there is a winning strategy of R for ψ. The problem of *synthesis* is then to construct such a strategy. The setting described here corresponds to the case where the system needs to satisfy a specification with respect to environments modeled by a rewrite system. Then, at each state, the system chooses the subset of transitions to proceed with and the environment provides the rules that determine the successors of the state.

Similar to Theorems 2.3.2 and 2.5.2 we construct automata that solve the realizability problem and provide winning strategies. The idea is simple: a strategy $f : Q \times V^* \to [m]$ can be viewed as a $V \times [m]$-labeled V-tree. Thus, the realizability problem can be viewed as the problem of determining whether we can augment the labels of the tree $\langle V^*, \tau_V \rangle$ by elements in $[m]$, and accept the augmented tree in a run of $\mathcal{A}$ in which whenever $\mathcal{A}$ reads an entry $i \in [m]$, it applies to the transition function of the specification graph automaton only rewrite rules in T_i.

We give the solution to the realizability and synthesis problems for branching-time specifications. Given a rewrite system R and a graph automaton $\mathcal{S}$, we show how to construct a 2APT $\mathcal{A}$ such that the language of $\mathcal{A}$ is not empty iff $\mathcal{S}$ is realizable over R.

Theorem 2.7.1 *Given a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$, a partition $\{T_1, \ldots, T_m\}$ of T, and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $((V \cup \{\bot\}) \times [m])$-labeled V-trees such that $L(\mathcal{A})$ contains exactly all the V-exhaustive trees whose projection on $[m]$ is a winning strategy of R for $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\| \cdot |V|)$ states, and its index is the index of $\mathcal{S}$ (plus 1 for a prefix-recognizable system).*

Proof: Unlike Theorem 2.3.2 here we use the emptiness problem of 2APT instead of the membership problem. It follows that we have to construct a 2APT that ensures that its input tree is V-exhaustive and that the strategy encoded in the tree is winning. The modification to the construction in the proof of Theorem 2.3.2 are simple. Let $\mathcal{A}'$ denote the result of the construction in Theorem 2.3.1 or Theorem 2.3.2 with the following modification to the function $apply_T$. From action states we allow to proceed only with transitions from T_i, where i is the $[m]$ element of the letter we read. For example, in the case of a pushdown system, we would have for $c \in \Delta$, $w \in W$, $q \in Q$, $A \in V$ and $i \in [m]$ (the new parameter to $apply_T$, which is read from the input tree),

$$apply_T(c, w, q, A, i) = \begin{cases} \langle \varepsilon, (w, q, \varepsilon) \rangle & \text{If } c = \varepsilon \\ \bigwedge_{\langle q, A, y, q' \rangle \in T_i} \langle \uparrow, (w, q', y) \rangle & \text{If } c = \Box \\ \bigvee_{\langle q, A, y, q' \rangle \in T_i} \langle \uparrow, (w, q', y) \rangle & \text{If } c = \Diamond \end{cases}$$

We now construct the automaton $\mathcal{A}'' = \langle (V \cup \{\bot\} \times [m]), (V \cup \{\bot\}), \rho, bot, \{V\}\rangle$ of index 1 (i.e., every valid run is an accepting run) such that for every $A, B \in V \cup \{\bot\}$ and $i \in [m]$ we have

$$\rho(A,(B,i)) = \begin{cases} \bigwedge_{C \in V}(C,C) & A = B \\ \textbf{false} & A \neq B \end{cases}$$

It follows that $\mathcal{A}'$ accepts only V-exhaustive trees. Finally, we take $\mathcal{A} = \mathcal{A}' \wedge \mathcal{A}''$ the conjunction of the two automata. □

Let $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$, let k be the index of $\mathcal{S}$, and let $\Gamma = (V \cup \{\bot\}) \times [m]$. By Theorem 2.2.5, we can transform $\mathcal{A}$ to a nondeterministic one-way parity tree automaton $\mathcal{N}$ with $2^{O(nk)}$ states and index $O(nk)$.[8] By [Rab69, Eme85], if $\mathcal{N}$ is nonempty, there exists a Γ-labeled V-tree $\langle V^*, f\rangle$ such that for all $\gamma \in \Gamma$, the set X_γ of nodes $x \in V^*$ for which $f(x) = \gamma$ is a regular set. Moreover, the nonemptiness algorithm of $\mathcal{N}$, which runs in time exponential in nk, can be easily extended to construct, within the same complexity, a deterministic word automaton $\mathcal{U}_\mathcal{A}$ over V such that each state of $\mathcal{U}_\mathcal{A}$ is labeled by a letter $\gamma \in \Gamma$, and for all $x \in V^*$, we have $f(x) = \gamma$ iff the state of $\mathcal{U}_\mathcal{A}$ that is reached by following the word x is labeled by γ. The automaton $\mathcal{U}_\mathcal{A}$ is then the answer to the synthesis problem. Note that since the transitions in $G_{\mathcal{R},f}$ take us from a state $x \in V^*$ to a state $y \in V^*$ such that x is not necessarily the parent of y in the V-tree, an application of the strategy f has to repeatedly run the automaton $\mathcal{U}_\mathcal{A}$ from its initial state resulting in a strategy whose every move is computed in time proportional to the length of the configuration. We can construct a strategy that computes the next step in time proportional to the difference between x and y. This strategy uses a pushdown store. It stores the run of $\mathcal{U}_\mathcal{A}$ on x on the pushdown store. In order compute the strategy in node y, we retain on the pushdown store only the part of the run of $\mathcal{U}_\mathcal{A}$ that relates to the common suffix of x and y. We then continue the run of $\mathcal{U}_\mathcal{A}$ on the prefix of y while storing it on the pushdown store.

The construction described in Theorems 2.3.1 and 2.3.2 implies that the realizability and synthesis problem is in EXPTIME. Thus, it is not harder than in the satisfiability problem for the μ-calculus, and it matches the known lower bound [FL79]. Formally, we have the following.

Theorem 2.7.2 *The realizability and synthesis problems for a pushdown or a prefix-recognizable rewrite system $\mathcal{R} = \langle \Sigma, V, Q, L, T, q_0, x_0\rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F\rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$, and k is the index of $\mathcal{S}$.*

By Theorem 2.2.7, if the specification is given by a μ-calculus formula ψ, the bound is the same, with $n = |\psi| \cdot |Q| \cdot \|T\| \cdot |V|$, and k being the alternation depth of ψ.

In order to use the above algorithm for realizability of linear-time specifications we cannot use the 'usual' translations of LTL to μ-calculus [Dam94, dAHM01]. The problem is with the fact that these translations are intended to be used in μ-calculus model checking. The translation from LTL to μ-calculus used for model checking [Dam94] cannot be used in the context of realizability [dAHM01]. We have to use a doubly exponential translation intended for realizability [dAHM01], this, however, results in a triple exponential algorithm which is, again, less than optimal.

[8] Notice that the automaton $\mathcal{A}''$ is in fact a 1NPT of index 1. We can improve the efficiency of the algorithm by first converting $\mathcal{A}'$ into a 1NPT and only then combining the result with $\mathcal{A}''$. This would result in $|V|$ being removed from the figure describing the index of $\mathcal{N}$.

Alur et al. show that LTL realizability and synthesis can be exponentially reduced to μ-calculus realizability [ATM03]. Given an LTL formula φ, they construct a graph automaton $\mathcal{S}_\varphi$ such that $\mathcal{S}_\varphi$ is realizable over R iff φ is realizable over R. The construction of the graph automaton proceeds as follows. According to Theorem 2.2.8, for every LTL formula ψ we can construct an NBW N_ψ such that $L(N_\psi) = L(\psi)$. We construct an NBW $N_{\neg\varphi} = \langle \Sigma, W, \eta, w_0, F\rangle$ from $\neg\varphi$. We then construct the graph automaton $\mathcal{S}_\varphi = \langle \Sigma, W, \rho, w_0, \{F, W\}\rangle$ where $\rho(w,\sigma) = \bigwedge_{w' \in \eta(w,\sigma)} \Box w'$ and the parity condition $\{F, W\}$ is equivalent to the co-Büchi condition F. It follows that $\mathcal{S}_\varphi$ is a universal automaton and has a unique run over every trace. Alur et al. show that the fact that $\mathcal{S}_\varphi$ has a unique run over every trace makes it adequate for solving the realizability of φ [ATM03]. The resulting algorithm is exponential in the rewrite system and doubly exponential in the LTL formula. As synthesis of LTL formulas with respect to finite-state environments is already 2EXPTIME-hard [PR89], this algorithm is optimal. Notice that realizability with respect to LTL specifications is exponential in the system already for pushdown systems and exponential in all components of the system for prefix-recognizable systems.

2.8 Discussion

The automata-theoretic approach has long been thought to be inapplicable for effective reasoning about infinite-state systems. We showed that infinite-state systems for which decidability is known can be described by finite-state automata, and therefore, the states and transitions of such systems can be viewed as nodes in an infinite tree and transitions between states can be expressed by finite-state automata. As a result, automata-theoretic techniques can be used to reason about such systems. In particular, we showed that various problems related to the analysis of such systems can be reduced to the membership or emptiness problems for alternating two-way tree automata. Our framework achieves the same complexity bounds of known model-checking algorithms and gives the first solution to model-checking LTL with respect to prefix-recognizable systems. We show that our framework also provides a solution to the realizability problem. In [PV04] we show how to extend it also to global model checking. In [Cac03, PV04] the scope of automata-theoretic reasoning is extended beyond prefix-recognizable systems.

We have shown that the problems of model checking with respect to pushdown systems with regular labeling and model checking with respect to prefix-recognizable systems are intimately related. We give reductions between model checking of pushdown systems with regular labeling and model checking of prefix-recognizable systems with simple labeling.

The automata-theoretic approach offers several extensions to the model checking setting. The systems we want to reason about are often augmented with *fairness constraints*. Like state properties, we can define a *regular fairness constraint* by a regular expression α, where a computation of the labeled transition graph is fair iff it contains infinitely many states in α (this corresponds to weak fairness; other types of fairness can be defined similarly). It is easy to extend our model-checking algorithm to handle fairness (that is, let the path quantification in the specification range only on fair paths[9]). In the branching-time framework, the automaton $\mathcal{A}$ can guess whether the state currently visited is in α, and then simulate the word automaton $\mathcal{U}_\alpha$ upwards, hoping to visit an accepting state when the root is reached. When $\mathcal{A}$ checks an existential property, it has to make sure that the property is satisfied along a fair path, and it is therefore required to visit infinitely

[9]The exact semantics of *fair graph automata* as well as *fair μ-calculus* is not straightforward, as they enable cycles in which we switch between existential and universal modalities. To make our point here, it is simpler to assume in the branching-time framework, say, graph automata that correspond to CTL* formulas.

many states in α. When $\mathcal{A}$ checks a universal property, it may guess that a path it follows is not fair, in which case $\mathcal{A}$ eventually always send copies that simulate the automaton for $\neg\alpha$. In the linear-time framework, we add the automata for the fairness constraints to the tree whose membership is checked. The guessed path violating the property must visit infinitely many fair states. The complexity of the model-checking algorithm stays the same.

Another extension is the treatment of μ-calculus specifications with *backwards modalities*. While forward modalities express weakest precondition, backward modalities express strongest postcondition, and they are very useful for reasoning about the past [LPZ85]. In order to adjust graph automata to backward reasoning, we add to Δ the "directions" $\Diamond^-$ and $\Box^-$. This enables the graph automata to move to predecessors of the current state. More formally, if a graph automaton reads a state x of the input graph, then fulfilling an atom $\Diamond^- t$ requires $\mathcal{S}$ to send a copy in state t to some predecessor of x, and dually for $\Box^- t$. Theorem 2.2.7 can then be extended to μ-calculus formulas and graph automata with both forward and backward modalities [Var98]. Extending our solution to graph automata with backward modalities is simple. Consider a configuration $(q, x) \in Q \times V^*$ in a prefix-recognizable graph. The predecessors of (q, x) are configurations $(q'y)$ for which there is a rule $\langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_{\gamma_i}$, z is accepted by $\mathcal{U}_{\beta_i}$, and y' is accepted by $\mathcal{U}_{\alpha_i}$. Hence, we can define a mapping T^- such that $\langle q, \gamma, \beta, \alpha, q'\rangle \in T^-$ iff $\langle q, \alpha, \beta, \gamma, q\rangle \in T$, and handle atoms $\Diamond^- t$ and $\Box^- t$ exactly as we handle $\Diamond t$ and $\Box t$, only that for them we apply the rewrite rules in T^- rather than these in T. The complexity of the model-checking algorithm stays the same. Note that the simple solution relies on the fact that the structure of the rewrite rules in a prefix-recognizable rewrite system is symmetric (that is, switching α and γ results in a well-structured rule), which is not the case for pushdown systems[10].

Recently, Alur et al. suggested the logic CARET, that can specify non-regular properties [AEM04]. Our algorithm generalizes to CARET specifications as well. Alur et al. show how to combine the specification with a pushdown system in a way that enables the application of our techniques. The logic CARET is tailored for use in conjunction with pushdown systems. It is not clear how to modify CARET in order to apply to prefix-recognizable systems.

Bibliography

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. 10th International Conference on Tools and Algorithms For the Construction and Analysis of Systems*, volume 2725 of *Lecture Notes in Computer Science*, pages 67–79, Barcelona, Spain, April 2004. Springer-Verlag.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. on Foundations of Computer Science*, pages 100–109, Florida, October 1997.

[ATM03] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive game graphs. In *Computer-Aided Verification, Proc. 15th International Conference*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2003.

[BC04] M. Bojanczyk and T. Colcombet. Tree-walking automata cannot be determinized. In *Proc. 31st International Colloquium on Automata, Languages, and Programing*, volume 3142 of *Lecture Notes in Computer Science*, pages 246–256. Springer-Verlag, 2004.

[10]Note that this does not mean we cannot model check specifications with backwards modalities in pushdown systems. It just means that doing so involves rewrite rules that are no longer pushdown. Indeed, a rule $\langle q, A, x, q'\rangle \in T$ in a pushdown system corresponds to the rule $\langle q, A, V^*, x, q'\rangle \in T$ in a prefix-recognizable system, inducing the rule $\langle q', x, V^*, A, q\rangle \in T^{-1}$.

[BCMS00] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. Unpublished manuscript, 2000.

[BE96] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1996.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, July 1997. Springer-Verlag.

[BLM01] P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessors using satisfiability solvers. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.

[BQ96] O. Burkart and Y.-M. Quemener. Model checking of infinite graphs defined by graph grammars. In *Proc. 1st International workshop on verification of infinite states systems*, volume 6 of *ENTCS*, page 15. Elsevier, 1996.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130, Stanford, CA, USA, August 2000. Springer-Verlag.

[BR01] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. 3rd Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 1992.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic J. Comut.*, 2:89–125, 1995.

[BS99] O. Burkart and B. Steffen. Model checking the full modal μ-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Bur97a] O. Burkart. Automatic verification of sequential infinite-state processes. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Lecture Notes in Computer Science*, volume 1354. Springer-Verlag, 1997.

[Bur97b] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. In F. Moller, editor, *Proc. 2nd International workshop on verification of infinite states systems*, 1997.

[Cac03] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *Proc. 30th International Colloqium on Automata, Languages, and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569, Eindhoven, The Netherlands, June 2003. Springer-Verlag.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 1996.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer-Verlag, 2001.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.

[CW02] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proc. 9th ACM conference on Computer and Communications Security*, pages 235–244, Washington, DC, USA, 2002. ACM.

[CW03] A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 2003.

[dAHM01] L. de Alfaro, T.A. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 279–290. IEEE Computer Society Press, 2001.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[DW99] M. Dickhfer and T. Wilke. Timed alternating tree automata: the automata-theoretic solution to the TCTL model checking problem. In *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 281–290, Prague, Czech Republic, 1999. Springer-Verlag, Berlin.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247, Chicago, IL, July 2000. Springer-Verlag.

[EHvB99] J. Engelfriet, H.J. Hoggeboom, and J.-P van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 328–337, White Plains, October 1988.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, San Juan, October 1991.

[EJS93] E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Comptuer Science*, pages 385–396, Elounda, Crete, June 1993. Springer-Verlag.

[EKS01] J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 316–339, Sendai, Japan, October 2001. Springer-Verlag.

[EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st Symp. on Logic in Computer Science*, pages 267–278, Cambridge, June 1986.

[Eme85] E.A. Emerson. Automata, tableaux, and temporal logics. In *Proc. Workshop on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 79–87. Springer-Verlag, 1985.

[Eme97] E.A. Emerson. Model checking and the μ-calculus. In N. Immerman and Ph.G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society, 1997.

[ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336, Paris, France, July 2001. Springer-Verlag.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown automata. In F. Moller, editor, *Proc. 2nd International Workshop on Verification of Infinite States Systems*, 1997.

[GO03] P. Gastin and D. Oddoux. LTL with past and two-Way very-Weak alternating automata. In *28th International Symposium on Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 439–448, Bratislava, Slovak Republic, August 2003. Springer-Verlag.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, May 1994.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996.

[HHWT95] T.A. Henzinger, P.-H Ho, and H. Wong-Toi. A user guide to HYTECH. In *Tools and algorithms for the construction and analysis of systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer-Verlag, 1995.

[HKV96] T.A. Henzinger, O. Kupferman, and M.Y. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In *Proc. 7th Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 514–529, Pisa, August 1996. Springer-Verlag.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *Proc. 20th International Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 552–562. Springer-Verlag, 1995.

[KNU03] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, Grenoble, France, April 2003. Springer-Verlag.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP95] O. Kupferman and A. Pnueli. Once and for all. In *Proc. 10th IEEE Symp. on Logic in Computer Science*, pages 25–35, San Diego, June 1995.

[KPV02] O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 2002.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[KV98] O. Kupferman and M.Y. Vardi. Modular model checking. In *Proc. Compositionality Workshop*, volume 1536 of *Lecture Notes in Computer Science*, pages 381–401. Springer-Verlag, 1998.

[KV99] O. Kupferman and M.Y. Vardi. Robust satisfaction. In *Proc. 10th Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 383–398. Springer-Verlag, August 1999.

[KV00a] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2000.

[KV00b] O. Kupferman and M.Y. Vardi. Synthesis with incomplete informatio. In *Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, January 2000.

[KV01] O. Kupferman and M.Y. Vardi. On bounded specifications. In *Proc. 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, pages 24–38. Springer-Verlag, 2001.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPY97] K. G. Larsen, P. Petterson, and W. Yi. UPPAAL: Status & developments. In *Computer Aided Verification, Proc. 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 456–459. Springer-Verlag, 1997.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, Brooklyn, June 1985. Springer-Verlag.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[Nev02] F. Neven. Automata, logic, and XML. In *16th International Workshop on Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26, Edinburgh, Scotland, September 2002. Springer-Verlag.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, Austin, January 1989.

[PV03] N. Piterman and M.Y. Vardi. From bidirectionality to alternation. *Theoretical Computer Science*, 295(1–3):295–321, February 2003.

[PV04] N. Piterman and M. Vardi. Global model-checking of infinite-state systems. In *Proc. 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–400. Springer-Verlag, 2004.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Sch02] S. Schwoon. *Model-checking pushdown systems.* PhD thesis, Technische Universität München, 2002.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th International Coll. on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer-Verlag, Berlin, July 1998.

[VB00] W. Visser and H. Barringer. Practical CTL* model checking: Should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer-Verlag, 1996.

[Wal00] I. Walukiewicz. Model checking ctl properties of pushdown systems. In *Proc. 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*, pages 127–138, New Delhi, India, December 2000. Springer-Verlag.

[Wil99] T. Wilke. CTL$^+$ is exponentially more succinct than CTL. In C. Pandu Ragan, V. Raman, and R. Ramanujam, editors, *Proc. 19th conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1999.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 185–194, Tucson, 1983.

[WW96] B. Willems and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Proc. 11th Symp. on Logic in Computer Science*, pages 294–303, New Brunswick, July 1996.

2.A Proof of Claim 2.4.4

The proof of the claim is essentially equivalent to the same proof in [PV03].

Claim 2.A.4 *$L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$.*

Proof: We prove that $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$ implies $L(\mathcal{A}) \neq \emptyset$. Let $r = (p_0, w_0) \cdot (p_1, w_1) \cdot (p_2, w_2) \cdots$ be an accepting run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$. We add the annotation of the locations in the run $(p_0, w_0, 0) \cdot (p_1, w_1, 1) \cdot (p_2, w_2, 2) \cdots$. We construct the run $\langle T', r' \rangle$ of $\mathcal{A}$. For every node $x \in T'$, if x is labeled by a singleton state we add a tag to x some triplet from the run r. If x is labeled by a pair state we add two tags to x, two triplets from the run r. The labeling and the tagging conform to the following.

- Given a node x labeled by state (p, d, α) and tagged by the triplet (p', w, i) from r, we build r' so that $p = p'$ and $d = \rho_\tau(w)$. Furthermore all triplets in r whose third element is greater than i have their second element greater or equal to w (Υ^* is ordered according to the lexical order on the reverse of the words).

- Given a node x labeled by state (q, p, d, α) and tagged by the triplets (q', w, i) and (p', w', j) from r, we build r' so that $q = q'$, $p = p'$, $w = w'$, $d = \rho_\tau(w)$, and $i < j$. Furthermore all triplets in r whose third element k is between i and j, have their second element greater or equal to w. Also, if $j > i + 1$ then $w_{j-1} = \upsilon \cdot w_j$ for some $\upsilon \in \Upsilon$.

Construct the run tree $\langle T', r' \rangle$ of $\mathcal{A}$ as follows. Label the root of T' by $(p_0, d_\tau^0, \bot)$ and tag it by $(p_0, \varepsilon, 0)$. Given a node $x \in T'$ labeled by (p, d, α) tagged by (p, w, i). Let (p_j, w_j, j) be the minimal $j > i$ such that $w_j = w$. If $j = i + 1$ then add one son to x, label it $(p_j, d, \bot)$ and tag it (p_j, w, j). If $j > i + 1$, then $w_{j-1} = \upsilon \cdot w_i$ for some $\upsilon \in \Upsilon$ and we add two sons to x, label them (p, p_j, d, β) and (p_j, d, β). We tag (p_i, p_j, d, β) by (p, w, i) and (p_j, w, j), and tag (p_j, d, β) by (p_j, w, j), β is $\top$ if there is a visit to F between locations i and j in r. If there is no other visit to w then $w_{i+1} = \upsilon \cdot w$ for some $\upsilon \in \Upsilon$. We add one son to x and label it $(p_{i+1}, \rho_\tau(d, \upsilon), \bot)$ and tag it $(p_{i+1}, \upsilon \cdot w, i + 1)$. Obviously the labeling and the tagging conform to the assumption.

Given a node x labeled by (p, q, d, α) tagged by (p, w, i) and (q, w, j). Let (p_k, w, k) be the first visit to w between i and j. If $k = i + 1$ then add one son to x, label it $(p_k, q, d, f_\alpha(p_k, q))$, and tag it by (p_k, w, k) and (q, w, j). If $k > i + 1$ then add two sons to x and label them $(p, p_k, d, f_{\beta_1}(p, p_k))$ and $(p_k, q, d, f_{\beta_2}(p_k, q))$ where β_1, β_2 are determined according to the visits to F between i and j. We tag $(p, p_k, d, f_{\beta_1}(p, p_k))$ by (p, w, i) and (p_k, w, k) and tag $(p_k, t, d, f_{\beta_2}(p_k, q))$ by (p_k, w, k) and (q, w', j).

If there is no visit to w between i and j it must be the case that all triplets in r between i and j have the same suffix $\upsilon \cdot w$ for some $\upsilon \in \Upsilon$ (otherwise w is visited). We add a son to x labeled $(p_{i+1}, q_{j-1}, \rho_\tau(d, \upsilon), f_\alpha(p', q'))$ and tagged by $(p_{i+1}, \upsilon \cdot w, i + 1)$ and $(p_{j-1}, \upsilon \cdot w, j - 1)$. We are ensured that $p_{j-1} \in C_q^{L_\tau(\rho_\tau(d,\upsilon))}$ as $(\uparrow, p_j) \in \delta(p_{j-1}, \tau(\upsilon \cdot w))$.

In the other direction, given an accepting run $\langle T', r' \rangle$ of $\mathcal{A}$ we use the recursive algorithm in Figure 2.1 to construct a run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$.

A node $x \cdot a$ in T' is *advancing* if the transition from x to $x \cdot a$ results from an atom $(1, r'(x \cdot a))$ that appears in $\eta(r'(x))$. An advancing node that is the immediate successor of a singleton state satisfies the disjunct $\bigvee_{\upsilon \in \Upsilon} \bigvee_{(\upsilon, p') \in \delta(p, L(d))} (1, (p', \rho_\tau(d, \upsilon), \bot))$ in η. We tag this node by the letter υ that was used to satisfy the transition. Similarly, an advancing node that is the immediate successor of a pair state satisfies the disjunct $\bigvee_{\upsilon \in \Upsilon} \bigvee_{\langle \upsilon, p' \rangle \in \delta(p_1, L_\tau(d))} \bigvee_{p'' \in C_{p_2}^{L_\tau(d)}} (1, (p', p'', \rho_\tau(d, \upsilon), f_\alpha(p', p''))$ in η. We tag this node by the letter υ that was used to satisfy the transition. We use these tags in order to build the run of $\mathcal{P}$. When handling advancing nodes we update the location on the tree Υ^* according to the tag. For an advancing node x we denote by $tag(x)$ the letter in Υ that tags it. A node is *non advancing* if the transition from x to $x \cdot a$ results from an atom $(0, r'(x \cdot a))$ that appears in $\eta(r'(x))$.

The function **build_run** uses the variable w to hold the location in the tree $\langle \Upsilon^*, \tau \rangle$. Working on a singleton (p, d, α) the variable add_l is used to determine whether p was already added to the run. Working on a pair (p, q, d, α) the variable add_l is used to determine whether p was already added to the run and the variable add_r is used to determine whether q was already added to the run.

The intuition behind the algorithm is quite simple. We start with a node x labeled by a singleton (p, d, α). If the node is advancing we update w by $tag(x)$. Now we add p to r (if needed). The case where x has one son matches a transition of the form $(\Delta, p') \in \delta(p, L_\tau(d))$. In this case we move to handle the son of x and clearly p' has to be added to the run r. In case $\Delta = \varepsilon$ the son of x is non advancing and p' reads the same location w. Otherwise, w is updated by Δ and p' reads $\Delta \cdot w$. The case where x has two sons matches a guess that there is another visit to w. Thus, the computation splits into two sons (p, q, d, β) and (q, d, β). Both sons are non advancing. The state p was already added to r and q is added to r only in the first son.

With a node x labeled by a pair (p, q, d, α), the situation is similar. The case where x has one non advancing son matches a transition of the form $(\epsilon, s') \in \delta(p, A)$. Then we move to the son. The state p' is added to r but q is not. The case where x has two non advancing sons matches a split to (p, p', d, α_1) and (p', q, d, α_2). Only p' is added to r as p and q are added by the current call to build_run or by an earlier call to build_run. The case where x has one advancing son matches the move to the state $(p', q', \rho_\tau(d, \upsilon), \alpha)$ and checking that $q' \in C_q^{L_\tau(\rho_\tau(d,\upsilon))}$. Both p' and q' are added to r and **handle**_C_q handles the sequence of ε transitions that connects q' to q.

It is quite simple to see that the resulting run is a valid and accepting run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$.

build_run $(x, r'(x) = (p, d, \alpha), w, add_l, add_r)$
  if (advancing(x))
    $w := tag(x) \cdot w$;
  if (add_l)
    $r := r \cdot (w, p)$;
  if (x has one son $x \cdot a$)
    build_run $(x \cdot a, r'(x \cdot a), w, 1, 0)$
  if (x has two sons $x \cdot a$ and $x \cdot b$)
    build_run $(x \cdot a, r'(x \cdot a), w, 0, 1)$
    build_run $(x \cdot b, r'(x \cdot b), w, 0, 0)$

handle_C_q $(r'(x) = (p', p, d, \alpha), q, w)$
  Let $t_0, \ldots, t_n \in P^+$ be the sequence of
  ε-transitions connecting p to q
    $r := r \cdot (w, t_1), \cdots, (w, t_{n-1})$

build_run $(x, r'(x) = (p, q, d, \alpha), w, add_l, add_r)$
  if (advancing(x))
    $w := tag(x) \cdot w$;
  if (add_l)
    $r := r \cdot (w, p)$;
  if (x has one non advancing son $x \cdot a$)
    build_run $(x \cdot a, r'(x \cdot a), w, 1, 0)$
  if (x has two sons $x \cdot a$ and $x \cdot b$)
    build_run $(x \cdot a, r'(x \cdot a), w, 0, 1)$
    build_run $(x \cdot b, r'(x \cdot b), w, 0, 0)$
  if (x has one advancing son $x \cdot a$)
    build_run $(x \cdot a, r'(x \cdot a), w, 1, 1)$
    handle_C_q $(r'(x \cdot a), q, tag(x \cdot a) \cdot w)$
  if (add_r)
    $r := r \cdot (w, q)$;

Figure 2.1: Converting a run of A into a run of P

□

2.B Lower Bound for Linear Time Model-Checking on Prefix-Recognizable Systems

It was shown by [BEM97] that the problem of model-checking an LTL formula with respect to a pushdown graph is EXPTIME-hard in the size of the formula. The problem is polynomial in the size of the pushdown system inducing the graph. Our algorithm for model-checking an LTL formula with respect to a prefix-recognizable graph is exponential both in the size of the formula and in $|Q_\beta|$.

As prefix-recognizable systems are a generalization of pushdown systems the exponential resulting from the formula cannot be improved. We show that also the exponent resulting from Q_β cannot be removed. We use the EXPTIME-hard problem of whether a linear space alternating Turing machine accepts the empty tape [CKS81]. We reduce this question to the problem of model-checking a fixed LTL formula with respect to the graph induced by a prefix-recognizable system with a constant number of states and transitions. Furthermore Q_α and Q_γ depend only on the alphabet of the Turing machine. The component Q_β does 'all the hard work'. Combining this with Theorem 2.5.2 we get the following.

Theorem 2.B.1 *The problem of model-checking the graph induced by the prefix-recognizable system* $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$ *is EXPTIME-complete in* $|Q_\beta|$.

Proof: Consider an alternating linear-space Turing machine $M = \langle \Gamma, S_u, S_e, \mapsto, s_0, F_{acc}, F_{rej} \rangle$. Let $f : \mathbb{N} \to \mathbb{N}$ be the linear function such that M uses $f(n)$ cells in its working tape in order to process an input of length n. In order to make sure that M does not accept the empty tape, we have to check that every legal pruning of the computation tree of M contains one rejecting branch.

Given an alternating linear-space Turing machine M as above, we construct a prefix-recognizable system R and an LTL formula φ such that $G_R \models \varphi$ iff M does not accept the empty tape. The system R has a constant number of states and rewrite rules. For every rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$, the languages of the regular expressions α_i and γ_i are subsets of $\Gamma \cup (\{\downarrow\} \times \Gamma) \cup S \cup \{\epsilon\}$. The language of the regular expression β_i, can be encoded by a nondeterministic automaton whose size is linear in n. The LTL formula φ does not depend on the structure of M.

The graph induced by R has one infinite trace. This trace searches for rejecting configurations in all the pruning trees. The trace first explores the left son of every configuration. If it reaches an accepting configuration, the trace backtracks until it reaches a universal configuration for which only the left son was explored. It then goes forward again and explores under the right son of the universal configuration. If the trace returns to the root without finding such a configuration then the currently explored pruning tree is accepting. Once a rejecting configuration is reached, the trace backtracks until it reaches an existential configuration for which only the left son was explored. It then explores under the right son of the existential configuration. In this mode, if the trace backtracks all the way to the root, it means that all pruning trees were checked and that there is no accepting pruning tree for M.

We change slightly the encoding of a configuration by including with the state of M a symbol l or r denoting whether the next explored configuration is the right or left successor. Let $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma \times \{l, r\})$ and let $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)} \cdot \sharp \sigma_1^d \ldots \sigma_{f(n)}^d$ be a configuration of M and its d-successor (where d is either l or r). We also set σ_0 and σ_0^d to $\sharp$. Given σ_{i-1}, σ_i, and σ_{i+1} we know, by the transition relation of M, what σ_i^d should be. In addition the symbol $\sharp$ should repeat exactly every $f(n) + 1$ letters. Let $next : V^3 \to V$ denote our expectation for σ_i^d. Notice, that whenever the triplet σ_{i-1}, σ_i, and σ_{i+1} does not include the reading head of the Turing machine, it does not matter whether d is l or r. In both cases the expectation for σ_i^d is the same. We set

$next(\sigma, \sharp, \sigma') = \sharp$, and

$$next(\sigma, \sigma', \sigma'') = \begin{cases} \sigma' & \{\sigma, \sigma', \sigma''\} \subseteq \{\sharp\} \cup \Gamma \\ \sigma' & \sigma'' = (s, \gamma, d) \text{ and } (s, \gamma) \rightarrow^d (s', \gamma', R) \\ (s', \sigma', d') & \sigma'' = (s, \gamma, d),\ (s, \gamma) \rightarrow^d (s', \gamma', L),\ \text{and } d' \in \{l, r\} \\ \sigma' & \sigma = (s, \gamma, d) \text{ and } (s, \gamma) \rightarrow^d (s', \gamma', L) \\ (s', \sigma', d') & \sigma = (s, \gamma, d),\ (s, \gamma) \rightarrow^d (s', \gamma', R),\ \text{and } d' \in \{l, r\} \\ \gamma' & \sigma' = (s, \gamma, d) \text{ and } (s, \gamma) \rightarrow^d (s', \gamma', \alpha) \end{cases}$$

Consistency with *next* now gives us a necessary condition for a sequence in V^* to encode a branch in the computation tree of M. Notice that when $next(\sigma, \sigma', \sigma'') \in S \times \Gamma \times \{l, r\}$ then marking it by both l and r is correct.

The prefix-recognizable system starts from the initial configuration of M. It has two main modes, a *forward* mode and a *backward* mode. In forward mode, the system guesses a new configuration. The configuration is guessed one letter at a time, and this letter should match the functions $next_l$ or $next_r$. If the computation reaches an accepting configuration, this means that the currently explored pruning tree might still be accepting. The system moves to backward mode and remembers that it should explore other universal branches until it finds a rejecting state. In backward universal mode, the system starts backtracking and removes configurations. Once it reaches a universal configuration that is marked by l, it replaces the mark by r, moves to forward mode, and explores the right son. If the root is reached (in backward universal mode), the computation enters a rejecting sink. If in forward mode, the system reaches a rejecting configuration, then the currently explored pruning tree is rejecting. The system moves to backward mode and remembers that it has to explore existential branches that were not explored. Hence, in backward existential mode, the system starts backtracking and removes configurations. Once it reaches an existential configuration that is marked by l, the mark is changed to r and the system returns to forward mode. If the root is reached (in backward existential mode) all pruning trees have been explored and found to be rejecting. Then the system enters an accepting sink. All that the LTL formula has to check is that there exists an infinite computation of the system and that it reaches the accepting sink. Note that the prefix-recognizable system accepts, when the alternating Turing machine rejects and vice versa.

More formally, the LTL formula is $\Diamond reject$ and the rewrite system is $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$ where

- $AP = \{reject\}$
- $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma \times \{l, r\})$
- $Q = \{forward, backward_\exists, backward_\forall, sink_a, sink_r\}$
- $L(q, \alpha) = \begin{cases} \emptyset & q \neq sink_a \\ \{reject\} & q = sink_a \end{cases}$
- $q_0 = forward$
- $x_0 = b \cdots b \cdot (s_0, b, l) \cdot \sharp$

In order to define the transition relation we use the following languages.

- $L^1_{egal} = \{next(\sigma, \sigma', \sigma'') \cdot V^{f(n)-1}\sigma \cdot \sigma' \cdot \sigma''\}$

  $L^2_{egal} = \{w \in V^{f(n)+1} \mid w \notin V^* \cdot \sharp \cdot V^* \cdot \sharp \cdot V^*\}$

  $L^3_{egal} = \{w \in V^{f(n)+1} \mid w \notin V^* \cdot (S \times \Gamma \times \{l, r\}) \cdot V^* \cdot (S \times \Gamma \times \{l, r\}) \cdot V^*\}$

  $L_{egal} = (L^1_{egal} \cap L^2_{egal} \cap L^3_{egal}) \cdot V^*$

  Thus, this language contains all words whose suffix of length $f(n)+1$ contains at most one $\sharp$ and at most one symbol from $S \times \Gamma \times \{l, r\}$ and the last letter is the *next* correct successor of the previous configuration.

- $A_{ccept} = V \cdot (\{F_{acc}\} \times \Gamma \times \{l, r\}) \cdot V^*$

  Thus, this language contains all words whose one before last letter is marked by an accepting state[11].

- $R_{eject} = V \cdot (\{F_{rej}\} \times \Gamma \times \{l, r\}) \cdot V^*$

  Thus, this language contains all words whose one before last letter is marked by a rejecting state.

- $R^{S_u \times \{l\}}_{emove} = V \setminus (S_u \times \Gamma \times \{l\})$

  Thus, this language contains all the letters that are not marked by universal states and the direction l.

- $R^{S_e \times \{l\}}_{emove} = V \setminus (S_e \times \Gamma \times \{l\})$.

  Thus, this language contains all the letters that are not marked by existential states and the direction l.

Clearly the languages L_{egal}, A_{ccept}, and R_{eject} can be accepted by nondeterministic automata whose size is linear in $f(n)$.

The transition relation includes the following rewrite rules:

1. $\langle forward, \{\epsilon\}, L_{egal}, V \setminus (S \times \Gamma \times \{r\}), forward\rangle$ - guess a new letter and put it on the store. States are guessed only with direction l. The fact that L_{egal} is used ensures that the currently guessed configuration (and in particular the previously guessed letter) is the successor of the previous configuration on the store.

2. $\langle forward, \{\epsilon\}, A_{ccept}, \{\epsilon\}, backward_\forall\rangle$ - reached an accepting configuration. Do not change the store and move to backward universal mode.

3. $\langle forward, \{\epsilon\}, R_{eject}, \{\epsilon\}, backward_\exists\rangle$ - reached a rejecting configuration. Do not change the store and move to backward existential mode.

4. $\langle backward_\forall, R^{S_u \times \{l\}}_{emove}, V^*, \{\epsilon\}, backward_\forall\rangle$ - remove one letter that is not in $S_u \times \Gamma \times \{l\}$ from the store.

[11] It is important to use the one before last letter so that the state itself is already checked to be the correct next successor of previous configuration.

5. $\langle backward_\forall, S_u \times \Gamma \times \{l\}, V^*, S_u \times \Gamma \times \{r\}, forward\rangle$ - replace the marking l by the marking r and move to forward mode. The state s does not change[12].

6. $\langle backward_\forall, \epsilon, \epsilon, \epsilon, sink_r\rangle$ - when the root is reached in backward universal mode enter the rejecting sink

7. $\langle backward_\exists, R_{remove}^{S_e \times \{l\}}, V^*, \{\epsilon\}, backward_\exists\rangle$ - remove one letter that is not in $S_e \times \Gamma \times \{l\}$ from the store.

8. $\langle backward_\exists, S_e \times \Gamma \times \{l\}, V^*, S_e \times \Gamma \times \{r\}, forward\rangle$ - replace the marking l by the marking r and move to forward mode. The state s does not change.

9. $\langle backward_\exists, \epsilon, \epsilon, \epsilon, sink_a\rangle$ - when the root is reached in backward existential mode enter the accepting sink.

10. $\langle sink_a, \epsilon, \epsilon, \epsilon, sink_a\rangle$ - remain in accepting sink.

11. $\langle sink_r, \epsilon, \epsilon, \epsilon, sink_r\rangle$ - remain in rejecting sink.

□

[12] Actually, we guess all states in S_u. As we change state into *forward*, the next transition verifies that indeed the state is the same state.

Chapter 3

Global model checking

We extend the automata-theoretic framework for reasoning about infinite-state sequential systems to handle also the global model-checking problem. Our framework is based on the observation that states of such systems, which carry a finite but unbounded amount of information, can be viewed as nodes in an infinite tree, and transitions between states can be simulated by finite-state automata. Checking that the system satisfies a temporal property can then be done by a two-way automaton that navigates through the tree. The framework is known for local model checking. For branching time properties, the framework uses two-way alternating automata. For linear time properties, the framework uses two-way path automata.

In order to solve the global model-checking problem we show that for both types of automata, given a regular tree, we can construct a nondeterministic word automaton that accepts all the nodes in the tree from which an accepting run of the automaton can start.

3.1 Introduction

An important research topic over the past decade has been the application of model checking to infinite-state systems. A major thrust of research in this area is the application of model checking to *infinite-state sequential systems*. These are systems in which a state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The origin of this thrust is the important result by Muller and Schupp that the monadic second-order theory of *context-free graphs* is decidable [MS85]. As the complexity involved in that decidability result is nonelementary, researchers sought decidability results of elementary complexity. This started with Burkart and Steffen, who developed an exponential-time algorithm for model-checking formulas in the *alternation-free* μ-calculus with respect to context-free graphs [BS92]. Researchers then went on to extend this result to the μ-calculus, on one hand, and to more general graphs on the other hand, such as *pushdown graphs* [BS95, Wal96], *regular graphs* [BQ96], and *prefix-recognizable graphs* [Cau96]. One of the most powerful results so far is an exponential-time algorithm by Burkart for model checking formulas of the μ-calculus with respect to prefix-recognizable graphs [Bur97b]. See also [BE96, BEM97, Bur97a, FWW97, BS99, BCMS00].[1] Some of this theory has also been reduced to practice. Pushdown model-checkers such as Mops [CW02], Moped [ES01, Sch02], and Bebop [BR00] (to name a few)

[1] Recently, it was shown that the monadic second-order theory of *high-order pushdown graphs* is decidable [KNU03]. This was adapted to solve μ-calculus model checking over such graphs, but the complexity of model-checking μ-calculus on a high order pushdown graph of level n is a stack of n exponentials [Cac03].

have been developed. Successful applications of these model-checkers to the verification of software are reported, for example, in [BR01, CW02].

We usually distinguish between *local* and *global* model checking. In the first setting we are given a specific state of the system and determine whether it satisfies a given property. In the second setting we compute (a finite representation of) the set of states that satisfy a given property. For many years global model-checking algorithms were the standard; in particular, CTL model checkers [CES86], and symbolic model checkers [BCM+92] perform global model checking. While local model checking holds the promise of reduced computational complexity [SW91] and is more natural for explicit LTL model checking [CVWY92], global model checking is especially important in cases where model checking is only part of the verification process. For example, in [CKV01, CKKV01], global model checking is used to supply coverage information, which informs us what parts of the design under verification are relevant to the specified properties. In [Sha00, LBBO01], an infinite-state system is abstracted into a finite-state system. Global model checking is performed over the finite-state system and the result is then used to compute invariants for the infinite-state system. In [PRZ01], results of global model checking over small instances of a parameterized system are generalized to invariants for every value of the system's parameter.

An automata-theoretic framework for reasoning about infinite-state sequential systems was developed in [KV00, KPV02] (see exposition in [Cac02a]). The automata-theoretic approach uses the theory of automata as a unifying paradigm for system specification, verification, and synthesis [WVS83, EJ91, Kur94, VW94, KVW00]. Automata enable the separation of the logical and the algorithmic aspects of reasoning about systems, yielding clean and asymptotically optimal algorithms. Traditionally automata-theoretic techniques provide algorithms only for local model checking [CVWY92, KV00, KPV02]. As model checking in the automata-theoretic approach is reduced to the emptiness of an automaton, it seems that this limitation to local model checking is inherent to the approach. For finite-state systems we can reduce global model checking to local model checking by iterating over all the states of the system, which is essentially what happens in symbolic model checking of LTL [BCM+92]. For infinite-state systems, however, such a reduction cannot be applied. In this paper we remove this limitation of automata-theoretic techniques. We show that the automata-theoretic approach to infinite-state sequential systems generalizes nicely to global model checking. Thus, all the advantages of using automata-theoretic methods, e.g., the ability to handle regular labeling and regular fairness constraints, the ability to handle μ-calculus with backward modalities, and the ability to check realizability [KV00, ATM03], apply also to the more general problem of global model checking.

We use two-way tree alternating automata to reason about properties of infinite-state sequential systems. The idea is based on the observation that states of such systems can be viewed as nodes in an infinite tree, and transitions between states can be simulated by finite-state automata. Checking that the system satisfies a temporal property can then be done by a two-way alternating automaton. Local model checking is then reduced to emptiness or membership problems for two-way tree automata

In this work, we give a solution to the global model-checking problem. The set of configurations of a prefix-recognizable system satisfying a μ-calculus property can be infinite, but it is regular, so it is finitely represented. We show how to construct a nondeterministic word automaton that accepts all the configurations of the system that satisfy (resp., do not satisfy) a branching-time (resp., linear-time) property. In order to do that, we study the *global membership* problem for two-way alternating parity tree automata and two-way path automata. Given a regular tree, the

global membership problem is to find the set of states of the automaton and locations on the tree from which the automaton accepts the tree. We show that in both cases the question is not harder than the simple membership problem (is the tree accepted from the root and the initial state). Our result matches the upper bounds for global model checking established in [BEM97, EHRS00, EKS01, KPV02, Cac02b]. Our contribution is in showing how this can be done uniformly in an automata-theoretic framework rather than via an eclectic collection of techniques.

3.2 Preliminaries

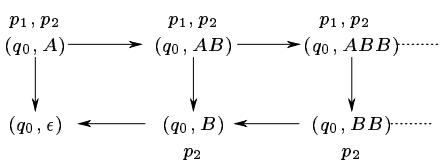
3.2.1 Labeled Rewrite Systems

A *labeled transition graph* is $G = \langle \Sigma, S, L, \rho, s_0 \rangle$, where Σ is a finite set of labels, S is a (possibly infinite) set of states, $L : S \to \Sigma$ is a labeling function, $\rho \subseteq S \times S$ is a transition relation, and $s_0 \in S_0$ is an initial state. When $\rho(s, s')$, we say that s' is a *successor* of s, and s is a *predecessor* of s'. For a state $s \in S$, we denote by $G^s = \langle \Sigma, S, L, \rho, s \rangle$, the graph G with s as its initial state. An *s-computation* is an infinite sequence of states $s_0, s_1, \ldots \in S^\omega$ such that $s_0 = s$ and for all $i \geq 0$, we have $\rho(s_i, s_{i+1})$. An s-computation $s_0, s_1, \ldots$ induces the *s-trace* $L(s_0) \cdot L(s_1) \cdots \in \Sigma^\omega$. Let $\mathcal{T}_s \subseteq \Sigma^\omega$ be the set of all s-traces.

A *rewrite system* is $R = \langle \Sigma, V, Q, L, T \rangle$, where Σ is a finite set of labels, V is a finite alphabet, Q is a finite set of states, $L : Q \times V^* \to \Sigma$ is a labeling function that depends only on the first letter of x (Thus, we may write $L : Q \times V \cup \{\epsilon\} \to \Sigma$. Note that the label is defined also for the case that x is the empty word ϵ). The finite set of rewrite rules T is defined below. The set of *configurations* of the system is $Q \times V^*$. Intuitively, the system has finitely many control states and an unbounded store. Thus, in a configuration $(q, x) \in Q \times V^*$ we refer to q as the *control state* and to x as the *store*. We consider here two types of rewrite systems. In a *pushdown* system, each rewrite rule is $\langle q, A, x, q' \rangle \in Q \times V \times V^* \times Q$. Thus, $T \subseteq Q \times V \times V^* \times Q$. In a *prefix-recognizable* system, each rewrite rule is $\langle q, \alpha, \beta, \gamma, q' \rangle \in Q \times reg(V) \times reg(V) \times reg(V) \times Q$, where $reg(V)$ is the set of regular expressions over V. Thus, $T \subseteq Q \times reg(V) \times reg(V) \times reg(V) \times Q$. For a word $w \in V^*$ and a regular expression $r \in reg(V)$ we write $w \in r$ to denote that w is in the language of the regular expression r. We note that the standard definition of prefix-recognizable systems does not include control states. Indeed, a prefix-recognizable system without states can simulate a prefix-recognizable system with states by having the state as the first letter of the unbounded store. We use prefix-recognizable systems with control states for the sake of uniform notation.

The rewrite system R starting in configuration (q_0, x_0) induces the labeled transition graph $G_R^{(q_0,x_0)} = \langle \Sigma, Q \times V^*, L', \rho_R, (q_0, x_0) \rangle$. States of G_R are the configurations of R and $\langle (q, z), (q', z') \rangle \in \rho_R$ if there is a rewrite rule $t \in T$ leading from configuration (q, z) to configuration (q', z'). Formally, if R is a pushdown system, then $\rho_R((q, A \cdot y), (q', x \cdot y))$ if $\langle q, A, x, q' \rangle \in T$; and if R is a prefix-recognizable system, then $\rho_R((q, x \cdot y), (q', x' \cdot y))$ if there are regular expressions α, β, and γ such that $x \in \alpha$, $y \in \beta$, $x' \in \gamma$, and $\langle q, \alpha, \beta, \gamma, q' \rangle \in T$. Note that in order to apply a rewrite rule in state $(q, z) \in Q \times V^*$ of a pushdown graph, we only need to match the state q and the first letter of z with the second element of a rule. On the other hand, in an application of a rewrite rule in a prefix-recognizable graph, we have to match the state q and we should find a partition of z to a prefix that belongs to the second element of the rule and a suffix that belongs to the third element. A labeled transition graph that is induced by a pushdown system is called a *pushdown graph*. A labeled transition system that is induced by a prefix-recognizable system is called a *prefix-recognizable graph*.

Example 3.2.1 *The pushdown system* $\langle 2^{\{p_1,p_2\}}, \{A,B\}, \{q_0\}, L, T\rangle$, *with* $T = \{ \langle q_0, A, AB, q_0 \rangle, \langle q_0, A, \varepsilon, q_0 \rangle, \langle q_0, B, \varepsilon, q_0 \rangle \}$, *and* $L(q_0, A) = \{p_1, p_2\}$, $L(q_0, B) = \{p_2\}$, *and* $L(q_0, \epsilon) = \emptyset$ *when starting from* (q_0, A) *induces the labeled transition graph on the right.*



Consider a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T\rangle$. For a rewrite rule $t_i = \langle s, \alpha_i, \beta_i, \gamma_i, s' \rangle \in T$, let $\mathcal{U}_\lambda = \langle V, Q_\lambda, q^0_\lambda, \eta_\lambda, F_\lambda\rangle$, for $\lambda \in \{\alpha_i, \beta_i, \gamma_i\}$, be the nondeterministic automaton for the language of the regular expression λ. We assume that all initial states have no incoming edges and that all accepting states have no outgoing edges. We collect all the states of all the automata for α, β, and γ regular expressions. Formally, $Q_\alpha = \bigcup_{t_i \in T} Q_{\alpha_i}$, $Q_\beta = \bigcup_{t_i \in T} Q_{\beta_i}$, and $Q_\gamma = \bigcup_{t_i \in T} Q_{\gamma_i}$.

We define the *size* $\|T\|$ of T as the space required in order to encode the rewrite rules in T and the labeling function. Thus, in a pushdown system, $\|T\| = \sum_{\langle q,A,x,q'\rangle \in T} |x|$, and in a prefix-recognizable system, $\|T\| = \sum_{\langle q,\alpha,\beta,\gamma,q'\rangle \in T} |\mathcal{U}_\alpha| + |\mathcal{U}_\beta| + |\mathcal{U}_\gamma|$.

We are interested in specifications expressed in the μ-calculus [Koz83] and in LTL [Pnu77]. For introduction to these logics we refer the reader to [Eme97]. We want to model check pushdown and prefix-recognizable systems with respect to specifications in these logics. We differentiate between *local* and *global* model-checking. In *local model-checking*, given a graph G and a specification φ, one has to determine whether G satisfies φ. In *global model-checking* we are interested in the set of configurations s such that G^s satisfies φ. As G is infinite, we hope to find a finite representation for this set. It is known that the set of configurations of a prefix-recognizable system satisfying a monadic second-order formula is regular [Cau96, Rab72], which implies that this also holds for pushdown systems and for μ-calculus and LTL specifications.

In this paper, we extend the automata-theoretic approach to model-checking of sequential infinite state systems [KV00, KPV02] to global model-checking. Our model-checking algorithm returns a nondeterministic finite automaton on words (NFW, for short) recognizing the set of configurations that satisfy (not satisfy, in the case of LTL) the specification. Our results match the previously known upper bounds [EHRS00, EKS01, Cac02b].[2]

Theorem 3.2.2 *Global model-checking for a system* R *and a specification* φ *is solvable*

- *in time* $(\|T\|)^3 \cdot 2^{O(|\varphi|)}$ *and space* $(\|T\|)^2 \cdot 2^{O(|\varphi|)}$, *where* R *is a pushdown system and* φ *is an LTL formula.*
- *in time* $(\|T\|)^3 \cdot 2^{O(|\varphi| \cdot |Q_\beta|)}$ *and space* $(\|T\|)^2 \cdot 2^{O(|\varphi| \cdot |Q_\beta|)}$, *where* R *is a prefix-recognizable system and* φ *is an LTL formula.*
- *in time* $2^{O(\|T\| \cdot |\varphi| \cdot k)}$, *where* R *is a prefix-recognizable system and* φ *is a* μ*-calculus formula of alternation depth* k.

[2] In order to obtain the stated bound for prefix-recognizable systems and LTL specifications one has to combine the result in [EKS01] with our reduction from prefix-recognizable systems to pushdown systems with regular labeling [KPV02].

3.2.2 Alternating Two-way Automata

Given a finite set Υ of directions, an Υ-*tree* is a set $T \subseteq \Upsilon^*$ such that if $v \cdot x \in T$, where $v \in \Upsilon$ and $x \in \Upsilon^*$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $v \in \Upsilon$ and $x \in T$, the node x is the *parent* of $v \cdot x$. Each node $x \neq \varepsilon$ of T has a *direction* in Υ. The direction of the root is the symbol $\perp$ (we assume that $\perp \notin \Upsilon$). The direction of a node $v \cdot x$ is v. We denote by $dir(x)$ the direction of node x. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $v \in \Upsilon$ such that $v \cdot x \in \pi$. Note that our definitions here dualize the standard definitions (e.g., when $\Upsilon = \{0, 1\}$, the successors of the node 0 are 00 and 10, rather than 00 and 01)[3].

Given two finite sets Υ and Σ, a Σ-*labeled* Υ-*tree* is a pair $\langle T, V \rangle$ where T is an Υ-tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, V \rangle$ a labeled tree. We say that an $((\Upsilon \cup \{\perp\}) \times \Sigma)$-labeled Υ-tree $\langle T, V \rangle$ is Υ-*exhaustive* if for every node $x \in T$, we have $V(x) \in \{dir(x)\} \times \Sigma$.

A tree is *regular* if it is the unwinding of some finite labeled graph. More formally, a *transducer* $\mathcal{D}$ is a tuple $\langle \Upsilon, \Sigma, Q, q_0, \eta, L \rangle$, where Υ is a finite set of directions, Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is a start state, $\eta : Q \times \Upsilon \to Q$ is a deterministic transition function, and $L : Q \to \Sigma$ is a labeling function. We define $\eta : \Upsilon^* \to Q$ in the standard way: $\eta(\varepsilon) = q_0$ and $\eta(ax) = \eta(\eta(x), a)$. Intuitively, a transducer is a labeled finite graph with a designated start node, where the edges are labeled by Υ and the nodes are labeled by Σ. A Σ-labeled Υ-tree $\langle \Upsilon^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{D} = \langle \Upsilon, \Sigma, Q, q_0, \eta, L \rangle$, such that for every $x \in \Upsilon^*$, we have $\tau(x) = L(\eta(x))$. We then say that the size of $\langle \Upsilon^*, \tau \rangle$, denoted $\|\tau\|$, is $|Q|$, the number of states of $\mathcal{D}$.

Alternating automata on infinite trees generalize nondeterministic tree automata and were first introduced in [MS87]. Here we describe alternating *two-way* tree automata. For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**, and, as usual, $\wedge$ has precedence over $\vee$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning **true** to elements in Y and assigning **false** to elements in $X \setminus Y$ makes θ true. For a set Υ of directions, the *extension* of Υ is the set $ext(\Upsilon) = \Upsilon \cup \{\varepsilon, \uparrow\}$ (we assume that $\Upsilon \cap \{\varepsilon, \uparrow\} = \emptyset$). An *alternating two-way automaton* over Σ-labeled Υ-trees is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \to \mathcal{B}^+(ext(\Upsilon) \times Q)$ is the transition function, and F specifies the acceptance condition.

A run of an alternating automaton $\mathcal{A}$ over a labeled tree $\langle \Upsilon^*, V \rangle$ is a labeled tree $\langle T_r, r \rangle$ in which every node is labeled by an element of $\Upsilon^* \times Q$. A node in T_r, labeled by (x, q), describes a copy of the automaton that is in the state q and reads the node x of Υ^*. Many nodes of T_r can correspond to the same node of Υ^*; there is no one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its successors have to satisfy the transition function. Formally, a run $\langle T_r, r \rangle$ is a Σ_r-labeled Γ-tree, for some set Γ of directions, where $\Sigma_r = \Upsilon^* \times Q$ and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\varepsilon, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, V(x)) = \theta$. Then there is a (possibly empty) set $S \subseteq ext(\Upsilon) \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, there is $\gamma \in \Gamma$ such that

[3] As will get clearer in the sequel, the reason for that is that rewrite rules refer to the prefix of words.

$\gamma \cdot y \in T_r$ and the following hold:

- If $c \in \Upsilon$, then $r(\gamma \cdot y) = (c \cdot x, q')$.
- If $c = \varepsilon$, then $r(\gamma \cdot y) = (x, q')$.
- If $c = \uparrow$, then $x = \upsilon \cdot z$, for some $\upsilon \in \Upsilon$ and $z \in \Upsilon^*$, and $r(\gamma \cdot y) = (z, q')$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $c = \uparrow$, we require that $x \neq \varepsilon$.

A run $\langle T_r, r \rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. We consider here *parity* acceptance conditions [EJ91]. A parity condition over a state set Q is a finite sequence $F = \{F_1, F_2, \ldots, F_m\}$ of subsets of Q, where $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_m = Q$. The number m of sets is called the *index* of $\mathcal{A}$. Given a run $\langle T_r, r \rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in \Upsilon^* \times \{q\}$. That is, $inf(\pi)$ is the set of states that appear infinitely often in π. A path π satisfies the condition F if there is an even i for which $inf(\pi) \cap F_i \neq \emptyset$ and $inf(\pi) \cap F_{i-1} = \emptyset$. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. The automaton $\mathcal{A}$ is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$. The Büchi acceptance condition [Büc62] is a private case of parity of index 3. The Büchi condition $F \subseteq Q$ is equivalent to the parity condition $\langle \emptyset, F, Q \rangle$. A path π satisfies the Büchi condition F iff $inf(\pi) \cap F \neq \emptyset$.

We say that $\mathcal{A}$ is one-way if δ is restricted to formulas in $B^+(\Upsilon \times Q)$. We say that $\mathcal{A}$ is nondeterministic if its transitions are of the form $\bigvee_{i \in I} \bigwedge_{\upsilon \in \Upsilon} (\upsilon, q_\upsilon^i))$, in such cases we write $\delta : Q \times \Sigma \to 2^{Q^{|\Upsilon|}}$. In the case that $|\Upsilon| = 1$, $\mathcal{A}$ is a word automaton.

Theorem 3.2.3 *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, we can construct an equivalent nondeterministic one-way parity tree automaton whose number of states is exponential in nk and whose index is linear in nk* [Var98]*, and we can check the nonemptiness of $\mathcal{A}$ in time exponential in nk* [EJS93].

The *membership problem* of an automaton $\mathcal{A}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$; or equivalently whether $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{A})$. For $q \in Q$ and $w \in \Upsilon^*$, we say that $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$ from (q, w) if there exists an accepting run of $\mathcal{A}$ that starts from state q reading node w (i.e. a run satisfying Condition 2 above where the root of the run tree is labeled by (w, q)). The *global membership problem* of $\mathcal{A}$ and regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine the set $\{(q, w) \mid \mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$ from $(q, w)\}$.

We use acronyms in $\{1, 2\} \times \{A, N\} \times \{B, P\} \times \{T, W\}$ to denote the different types of automata. The first symbol stands for the type of movement of the automaton: 1 for 1-way automata (we often omit the 1) and 2 for 2-way automata. The second symbol stands for the branching mode of the automaton: A for alternating and N for nondeterministic. The third symbol stands for the type of acceptance used by the automaton: B for Büchi and P for parity, and the last symbol stands for the object the automaton is reading: W for words and T for trees. For example, a 2APT is a 2-way alternating parity tree automaton and an NBW is a 1-way nondeterministic Büchi word automaton.

3.2.3 Alternating Automata on Labeled Transition Graphs

Consider a labeled transition graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$. Let $\Delta = \{\varepsilon, \Box, \Diamond\}$. An alternating automaton on labeled transition graphs (*graph automaton*, for short) [Wil99] [4] is a tuple $\mathcal{S} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where Σ, Q, q_0, and F are as in alternating two-way automata, and $\delta : Q \times \Sigma \to \mathcal{B}^+(\Delta \times Q)$ is the transition function. Intuitively, when $\mathcal{S}$ is in state q and it reads a state s of G, fulfilling an atom $\langle \Diamond, t \rangle$ (or $\Diamond t$, for short) requires $\mathcal{S}$ to send a copy in state t to some successor of s. Similarly, fulfilling an atom $\Box t$ requires $\mathcal{S}$ to send copies in state t to all the successors of s. Thus, graph automata cannot distinguish between the various successors of a state and treat them in an existential or universal way.

Like runs of alternating two-way automata, a run of a graph automaton $\mathcal{S}$ over a labeled transition graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$ is a labeled tree in which every node is labeled by an element of $S \times Q$. A node labeled by (s, q), describes a copy of the automaton that is in the state q of $\mathcal{S}$ and reads the state s of G. Formally, a run is a Σ_r-labeled Γ-tree $\langle T_r, r \rangle$, where Γ is some set of directions, $\Sigma_r = S \times Q$, and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (s_0, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (s, q)$ and $\delta(q, L(s)) = \theta$. Then there is a (possibly empty) set $S \subseteq \Delta \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, we have:
   - If $c = \varepsilon$, then there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s, q')$.
   - If $c = \Box$, then for every successor s' of s, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.
   - If $c = \Diamond$, then there is a successor s' of s and $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.

Acceptance is defined as in 2APT runs. The graph G is accepted by $\mathcal{S}$ if there is an accepting run on it. We denote by $\mathcal{L}(\mathcal{S})$ the set of all graphs that $\mathcal{S}$ accepts and by $\mathcal{S}^q = \langle \Sigma, Q, q, \delta, F \rangle$ the automaton $\mathcal{S}$ with q as its initial state.

We use graph automata as our branching time specification language. We say that a labeled transition graph G satisfies a graph automaton $\mathcal{S}$, denoted $G \models \mathcal{S}$, if $\mathcal{S}$ accepts G. Graph automata have the same expressive power as the μ-calculus. Formally,

Theorem 3.2.4 [Wil99] *Given a μ-calculus formula ψ, of length n and alternation depth k, we can construct a graph parity automaton $\mathcal{S}_\psi$ such that $\mathcal{L}(\mathcal{S}_\psi)$ is exactly the set of graphs satisfying ψ. The automaton $\mathcal{S}_\psi$ has n states and index k.*

We use NBW as our linear time specification language. We say that a labeled transition graph G satisfies an NBW N, denoted $G \models N$, if $\mathcal{T}_{s_0} \cap L(N) \neq \emptyset$ (where s_0 is the initial state of G)[5]. We are especially interested in cases where $\Sigma = 2^{AP}$, for some set AP of atomic propositions AP, and in languages $L \subseteq (2^{AP})^\omega$ definable by NBW or formulas of the linear temporal logic LTL [Pnu77]. For an LTL formula φ, the *language* of φ, denoted $L(\varphi)$, is the set of infinite words that satisfy φ.

Theorem 3.2.5 [VW94] *For every LTL formula φ, we can construct an NBW N_φ with $2^{O(|\varphi|)}$ states such that $L(N_\varphi) = L(\varphi)$.*

[4] See related formalism in [JW95].

[5] Notice, that our definition dualizes the usual definition for LTL. Here, we say that a linear time specification is satisfied if there exists a trace that satisfies it. Usually, a linear time specification is satisfied it if all traces satisfy it.

Given a graph G and a specification $\mathcal{S}$, the *global model-checking* problem is to compute the set of configurations s of G such that $G^s \models \mathcal{S}$. Whether we are interested in branching or linear time model-checking is determined by the type of automaton used.

3.3 Global Membership for 2APT

In this section we solve the global membership problem for 2APT. Let $\mathcal{A} = \langle \Sigma, S, s_0, \rho, \alpha \rangle$ be a 2APT and $T = \langle \Upsilon^*, \tau \rangle$ a regular tree. Our construction consists of two stages. First, we modify $\mathcal{A}$ into a 2APT $\mathcal{A}'$ that starts its run from the root of the tree in an idle state. In this idle state it goes to a node in the tree that is marked with a state of $\mathcal{A}$. From that node, the new automaton starts a fresh run of $\mathcal{A}$ from the marked state. We convert $\mathcal{A}'$ into an NPT $\mathcal{P}$ [Var98]. Second, we combine $\mathcal{P}$ with an NBT $\mathcal{D}'$ that accepts only trees that are identical to the regular tree T and in addition have exactly one node marked by some state of $\mathcal{A}$. We check now the emptiness of this automaton $\mathcal{A}''$. From the emptiness information we derive an NFW N that accepts a word $w \in \Upsilon^*$ in state $s \in S$ (i.e. the run ends in state s of $\mathcal{A}$; state s is an accepting state of N) iff $\mathcal{A}$ accepts T from (s, w).

Theorem 3.3.1 *Consider a 2APT $\mathcal{A} = \langle \Sigma, S, s_0, \rho, \alpha \rangle$ and a regular tree $T = \langle \Upsilon^*, \tau \rangle$. We can construct an NFW $N = \langle \Upsilon, R' \cup S, r_0, \Delta, S \rangle$ that accepts the word w in state $s \in S$ iff $\mathcal{A}$ accepts T from (s, w). Let n be the number of states of $\mathcal{A}$ and h its index; the NFW N is constructible in time exponential in nh and polynomial in $||\tau||$.*

Proof: Let $S_+ = S \cup \{\bot\}$ and $\Upsilon = \{\upsilon_1, \ldots, \upsilon_k\}$. Consider the 2APT $\mathcal{A}' = \langle \Sigma \times S_+, S', s'_0, \rho', \alpha' \rangle$ where $S' = S \cup \{s'_0\}$, s'_0 is a new initial state, α' is identical to α except having s'_0 belonging to some odd set of α', and ρ' is defined as follows.

$$\rho'(s, (\sigma, t)) = \begin{cases} \rho(s, \sigma) & s \neq s'_0 \\ \bigvee_{\upsilon \in \Upsilon} (\upsilon, s'_0) & s = s'_0 \text{ and } t = \bot \\ \bigvee_{\upsilon \in \Upsilon} (\upsilon, s'_0) \vee (\varepsilon, s') & s = s'_0 \text{ and } t = s' \end{cases}$$

Clearly, $\mathcal{A}'$ accepts a $(\Sigma \times S_+)$-labeled tree T' iff there is a node x in T' labeled by (σ, s) for some $(\sigma, s) \in \Sigma \times S$ and $\mathcal{A}$ accepts the projection of T' on Σ when it starts its run from node x in state s. Let $\mathcal{P} = \langle \Sigma \times \mathcal{S}_+, P, p_0, \rho_1, \alpha_1 \rangle$ be the NPT that accepts exactly those trees accepted by $\mathcal{A}'$ [Var98]. If $\mathcal{A}$ has n states and index h then $\mathcal{P}$ has $(nh)^{O(nh)}$ states and index $O(nh)$.

Let $\mathcal{D} = \langle \Upsilon, \Sigma, Q, q_0, \eta, L \rangle$ be the transducer inducing the labeling τ of T. We construct an NBT $\mathcal{D}'$ that accepts $(\Sigma \times S_+)$-labeled trees whose projection on Σ is τ and have exactly one node marked by a state in S. Consider the NBT $\mathcal{D}' = \langle \Sigma \times S_+, Q \times \{\bot, \top\}, (q_0, \bot), \eta', Q \times \{\top\} \rangle$ where η' is defined as follows. For $q \in Q$ let $pend_i(q) = \langle (\eta(q, \upsilon_1), \top), \ldots, (\eta(q, \upsilon_i), \bot), \ldots, (\eta(q, \upsilon_k), \top) \rangle$ be the tuple where the j-th element is the υ_j-successor of q and all elements are marked by $\top$ except for the i-th element, which is marked by $\bot$. Intuitively, a state $(q, \top)$ accepts a subtree all of whose nodes are marked by $\bot$. A state $(q, \bot)$ means that D' is still searching for the unique node labeled by a state in S. The transition to $pend_i$ means that D' is looking for that node in direction $\upsilon_i \in \Upsilon$.

$$\eta'((q, \beta), (\sigma, \gamma)) = \begin{cases} \{\langle (\eta(q, \upsilon_1), \top), \ldots, (\eta(q, \upsilon_k), \top) \rangle\} & \beta = \top,\ \gamma = \bot \text{ and } \sigma = L(q) \\ \{\langle (\eta(q, \upsilon_1), \top), \ldots, (\eta(q, \upsilon_k), \top) \rangle\} & \beta = \bot,\ \gamma \in S \text{ and } \sigma = L(q) \\ \{pend_i(q) \mid i \in [1..k]\} & \beta = \gamma = \bot \text{ and } \sigma = L(q) \\ \emptyset & \text{Otherwise} \end{cases}$$

Clearly, $\mathcal{D}'$ accepts a $(\Sigma \times S_+)$-labeled tree T' iff the projection of T' on Σ is exactly τ and all nodes of T' are labeled by $\perp$ except one node labeled by some state $s \in S$.

Let $\mathcal{A}'' = \langle \Sigma \times S_+, R, r_0, \delta, \alpha_2 \rangle$ be the product of $\mathcal{D}'$ and $\mathcal{P}$ where $R = (Q \times \{\perp, \top\}) \times P$, $r_0 = ((q_0, \perp), p_0)$, δ is defined below and $\alpha_2 = \langle F'_1, \ldots, F'_m \rangle$ is obtained from $\alpha_1 = \langle F_1, \ldots, F_m \rangle$ by setting $F'_1 = ((Q \times \{\perp, \top\}) \times F_1) \cup (Q \times \{\perp\} \times P)$ and for $i > 1$ we have $F'_i = (Q \times \{\top\}) \times F_i$. Thus, $\perp$ states are visited finitely often, and otherwise only the state of $\mathcal{P}$ is important for acceptance. For every state $((q, \beta), p) \in (Q \times \{\perp, \top\}) \times P$ and letter $(\sigma, \gamma) \in \Sigma \times S_+$ the transition function δ is defined by:

$$\delta \quad (((q, \beta), p), (\sigma, \gamma)) = \left\{ \langle ((q_1, \beta_1), p_1), \ldots, ((q_k, \beta_k), p_k) \rangle \;\middle|\; \begin{array}{c} \langle p_1, \ldots, p_k \rangle \in \rho_1(p, (\sigma, \gamma)) \text{ and} \\ \langle (q_1, \beta_1), \ldots, (q_k, \beta_k) \rangle \in \eta'((q, \beta), (\sigma, \gamma)) \end{array} \right\}$$

Every tree T' accepted by $\mathcal{A}''$ has a unique node x labeled by a state s of $\mathcal{A}$ and all other nodes are labeled by $\perp$, and if T is the projection of T' on Σ then $\mathcal{A}$ accepts T from (s, x).

The number of states of $\mathcal{A}''$ is $\|\tau\| \cdot (nh)^{O(nh)}$ and its index is $O(nh)$. We can check whether $\mathcal{A}''$ accepts the empty language in time exponential in nh. The emptiness algorithm returns the set of states of $\mathcal{A}''$ whose language is not empty [EJS93]. From now on we remove from the state space of $\mathcal{A}''$ all states whose language is empty. Thus, transitions of $\mathcal{A}''$ contain only tuples such that all states in the tuple have non empty language.

We are ready to construct the NFW N. The states of N are the states of $\mathcal{A}''$ in $(Q \times \{\perp\}) \times P$ in addition to S (the set of states of $\mathcal{A}$). Every state in S is an accepting sink of N. For the transition of N we follow transitions of $\perp$-states. Once we can transition into a tuple where the $\perp$ is removed, we transition into the appropriate accepting states.

Let $N = \langle \Upsilon, R' \cup S, r_0, \Delta, S \rangle$, where $R' = R \cap (Q \times \{\perp\} \times P)$, r_0 is the initial state of $\mathcal{A}''$, S is the set of states of $\mathcal{A}$ (accepting sinks in N), and Δ is defined below.

Consider a state $((q, \perp), p) \in R'$. Its transition in $\mathcal{A}''$ is of the form

$$\delta \quad (((q, \perp), p), (L(q), \perp)) = \left\{ \langle ((q_1, \top), p_1), \ldots, ((q_i, \perp), p_i), \ldots, ((q_k, \top), p_k) \rangle \;\middle|\; \begin{array}{c} q_j = \eta(q, \upsilon_j) \text{ and} \\ \langle p_1, \ldots, p_k \rangle \in \rho_1(p, (L(q), \perp)) \end{array} \right\}$$

$$\delta \quad (((q, \perp), p), (L(q), s)) = \left\{ \langle ((q_1, \top), p_1), \ldots, ((q_k, \top), p_k) \rangle \;\middle|\; \begin{array}{c} q_j = \eta(q, \Upsilon_j) \text{ and} \\ \langle p_1, \ldots, p_k \rangle \in \rho_1(p, (L(q), \perp)) \end{array} \right\}$$

For every tuple $\langle ((q_1, \top), p_1), \ldots, ((q_i, \perp), p_i), \ldots, ((q_k, \top), p_k) \rangle$appearing in $\delta(((q, \perp), p), (L(q), \perp))$, we add $((q_i, \perp), p_i)$ to $\Delta(((q, \perp), p), \upsilon_i)$. For every tuple $\langle ((q_1, \top), p_1), \ldots, ((q_k, \top), p_k) \rangle$appearing in $\delta(((q, \perp), p), (L(q), s))$, we add the letter s to $\Delta(((q, \perp), p), \epsilon)$.

Lemma 3.3.2 *A word $w \in \Upsilon^*$ is accepted by N in a state $s \in S$ iff $\mathcal{A}$ accepts T from (w, s).*

Proof: Given a node $w \in \Upsilon^*$ and a state $s \in S$ let the tree T^s_w be the unique $(\Sigma \times S_+)$-labeled tree whose projection on Σ is T and its unique node labeled by a state in S is w that is labeled by s.

Suppose that N accepts w with the run $r = ((q_0, \perp), p_0), \ldots, ((q_n, \perp), p_n), s$ that ends in s. We construct an accepting run tree $r' : \Upsilon^* \to R$ of $\mathcal{A}''$ on T^s_w. Let $r'(\epsilon) = ((q_0, \perp), p_0)$. Clearly, $r'(\epsilon) = r_0$. Continue by induction the run r' from a node $x \in \Upsilon^*$ labeled by $((q_i, \perp), p_i)$. From the

definition of N it follows that for every two adjacent states in r, $((q_i, \bot), p_i), ((q_{i+1}, \bot), p_{i+1})$ the transition $\delta(((q_i, \bot), p_i), (\sigma, \bot))$ of $\mathcal{A}''$ contains a tuple $\langle$ ($(q_1^{i+1}, \top)$, p_1^{i+1}), ..., $((q_j^{i+1}, \bot), p_j^{i+1})$, ..., $((q_k^{i+1}, \top), p_k^{i+1})$ $\rangle$ such that $\sigma = L(q_i)$, $q_j^{i+1} = q_{i+1}$, $p_j^{i+1} = p_{i+1}$ and for every l we have that the language of $((q_l^{i+1}, \alpha), p_l^{i+1})$ is not empty. For $l \neq j$ we add some accepting run tree of $((q_l^{i+1}, \top), p_l^{i+1})$ under $x \cdot \upsilon_l$. We label $x \cdot \upsilon_j$ by $((q_{i+1}, \bot), p_{i+1})$. Similarly, when we reach the end of the run of N, the transition $\delta(((q_n, \bot), p_n), (L(q), s))$ contains a tuple $\langle$ $((q_1^{n+1}, \top), p_1^{n+1})$, ..., $((q_k^{n+1}, \top), p_k^{n+1})$ $\rangle$ such that for every state in the tuple its language is not empty. We now add a complete accepting run tree below every successor of the node x and complete the accepting run r'' of $\mathcal{A}''$. It follows from the definition of $\mathcal{A}''$ that $\mathcal{A}$ accepts T from (s, w).

Suppose that $\mathcal{A}$ accepts T from (s, w) then we conclude that $\mathcal{A}''$ accepts T_w^s and from the accepting run of $\mathcal{A}''$ we construct an accepting run of N on w that ends in state s. ▟

□

3.4 Global Model Checking of Branching Time Properties

In this section we solve the global model-checking for branching time specifications by a reduction to the global membership problem for 2APT. We start by demonstrating our technique on global model-checking for pushdown systems. Then we show how to extend it to prefix-recognizable systems. The construction is somewhat different from the construction in [KV00] as we use the global-membership of 2APT instead of the emptiness of 2APT.

Consider a rewrite system $R = \langle \Sigma, V, Q, L, T \rangle$. Recall that a configuration of R is a pair $(q, x) \in Q \times V^*$. Thus, the store x corresponds to a node in the full infinite V-tree. An automaton that reads the tree V^* can memorize in its state space the state component of the configuration and refer to the location of its reading head in V^* as the store. We would like the automaton to "know" the location of its reading head in V^*. A straightforward way to do so is to label a node $x \in V^*$ by x. This, however, involves an infinite alphabet. We show that labeling every node in V^* by its direction is sufficiently informative to provide the 2APT with the information it needs in order to simulate transitions of the rewrite system. Let $\langle V^*, \tau_V \rangle$ be the tree where $\tau_V(x) = dir(x)$.

3.4.1 Pushdown Systems

In this section we present our solution for pushdown systems in details.

Theorem 3.4.1 *Given a pushdown system $R = \langle \Sigma, V, Q, L, T \rangle$ and a graph automaton $\mathcal{W} = \langle \Sigma, W, w_0, \delta, F \rangle$, we can construct a 2APT $\mathcal{A}$ on V-trees and a function f that associates states of $\mathcal{A}$ with states of R such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ from (p, x) iff $G_R^{(f(p),x)} \models \mathcal{W}$. The automaton $\mathcal{A}$ has $O(|Q| \cdot \|T\| \cdot |V|)$ states, and has the same index as $\mathcal{W}$.*

Proof: Let $V = \{A_1, \ldots, A_n\}$ and $\Sigma = V \cup \{\bot\}$. Recall that in order to apply a rewrite rule of a pushdown system from configuration (q, x), it is sufficient to know q and the first letter of x. Let $\langle V^*, \tau_V \rangle$ be the V-labeled V-tree such that for every $x \in V^*$ we have $\tau_V(x) = dir(x)$. Note that $\langle V^*, \tau_V \rangle$ is a regular tree of size $|V| + 1$. We define $\mathcal{A} = \langle V, P, \eta, p_0, \alpha \rangle$ as follows.

- $P = (W \times Q \times \mathit{tails}(T))$, where $\mathit{tails}(T) \subseteq V^*$ is the set of all suffixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{A}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that G_R with initial state $(q, y \cdot x)$ is accepted by $\mathcal{W}^w$. In particular, when $y = \varepsilon$, then G_R with initial state (q, x) (the node currently being visited) needs to be accepted by $\mathcal{W}^s$. States of the form $\langle w, q, \varepsilon \rangle$ are called *action states*. From these states $\mathcal{A}$ consults δ and T in order to impose new requirements on the exhaustive V-tree. States of the form $\langle w, q, y \rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{A}$ only navigates downwards y to reach new action states.

- In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(\mathit{ext}(V) \times P)$, we first define the function $\mathit{apply}_T : \Delta \times W \times Q \times V \to \mathcal{B}^+(\mathit{ext}(V) \times P)$. Intuitively, apply_T transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words of the form $A \cdot V^*$. For $c \in \Delta$, $w \in W$, $q \in Q$, and $A \in V$ we define

$$\mathit{apply}_R(c, w, q, A) = \begin{cases} \langle \varepsilon, (w, q, \varepsilon) \rangle & \text{If } c = \varepsilon \\ \bigwedge_{\langle q, A, y, q' \rangle \in T} \langle \uparrow, (w, q', y) \rangle & \text{If } c = \Box \\ \bigvee_{\langle q, A, y, q' \rangle \in T} \langle \uparrow, (w, q', y) \rangle & \text{If } c = \Diamond \end{cases}$$

  Note that T may contain no tuples in $\{q\} \times \{A\} \times V^* \times Q$ (that is, the transition relation of G_R may not be total). In particular, this happens when $A = \bot$ (that is, for every state $q \in Q$ the configuration the state (q, ε) of G_R has no successors). Then, we take empty conjunctions as **true**, and take empty disjunctions as **false**.

  In order to understand the function apply_R, consider the case $c = \Box$. When $\mathcal{W}$ reads the configuration $(q, A \cdot x)$ of the input graph, fulfilling the atom $\Box s$ requires $\mathcal{S}$ to send copies in state w to all the successors of $(q, A \cdot x)$. The automaton $\mathcal{A}$ then sends to the node x copies that check whether all the configuration $(q', y \cdot x)$, with $\rho_R((q, A \cdot x), (q', y \cdot x))$, are accepted by $\mathcal{W}$ with initial state w.

  Now, for a formula $\theta \in \mathcal{B}^+(\Delta \times W)$, the formula $\mathit{apply}_R(\theta, q, A) \in \mathcal{B}^+(\mathit{ext}(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w \rangle$ by the atom $\mathit{apply}_R(c, w, q, A)$. We can now define η for all $A \in V \cup \{\bot\}$ as follow.

  - $\eta(\langle w, q, \varepsilon \rangle, A) = \mathit{apply}_R(\delta(w, L(q, A)), q, A)$.
  - $\eta(\langle w, q, B \cdot y \rangle, A) = (B, \langle w, q, y \rangle)$.

  Thus, in action states, $\mathcal{A}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to δ. In navigation states, $\mathcal{A}$ needs to go downwards $B \cdot y$.

- F' is obtained from F by replacing each set F_i by the set $F_i \times Q \times \mathit{tails}(R)$.

The function f associates with state (w, q, ϵ) the state q of R. For other states, f is undefined. $\Box$

Pushdown rewrite systems are a special case of prefix-recognizable rewrite systems. In the next section we describe how to extend the construction described above to prefix-recognizable systems, and we also analyze the complexity of the model-checking algorithm that follows for the two types of systems.

3.4.2 Prefix-Recognizable Systems

In this section we extend the construction described above to prefix-recognizable transition systems.Again the two-way automaton navigates through the full V-tree and simulates transitions of the rewrite system. In order to apply a rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$, the automaton goes up the tree along a word in α_i, it checks that the suffix is in β_i by sending a separate copy to the root, and moves downwards along a word in γ_i.

Theorem 3.4.2 *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T \rangle$ and a graph automaton $\mathcal{W} = \langle \Sigma, W, w_0, \delta, F \rangle$, we can construct a 2APT $\mathcal{A}$ on V-trees and a function f that associates states of $\mathcal{A}$ with states of R such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ from (p, x) iff $G_R^{(f(p),x)} \models \mathcal{W}$. The automaton $\mathcal{A}$ has $O(|Q| \cdot \|T\| \cdot |V|)$ states, and has the same index as $\mathcal{W}$.*

Proof: Let $Q_\Omega = Q_\alpha \cup Q_\beta \cup Q_\gamma$ be the union of all the state spaces of the automata associated with regular expressions that participate in T.

As in the case of pushdown systems, $\mathcal{A}$ uses the labels of the tree to learn the state in V^* that each node corresponds to. As there, $\mathcal{A}$ applies to the transition function δ of $\mathcal{W}$ the rewrite rules of R. Here, however, the application of the rewrite rules on atoms of the form $\Diamond w$ and $\Box w$ is more involved, and we describe it below. Assume that $\mathcal{A}$ wants to check whether $\mathcal{W}^w$ accepts $G_R^{(q,x)}$, and it wants to proceed with an atom $\Diamond w'$ in $\delta(w)$. The automaton $\mathcal{A}$ needs to check whether $\mathcal{W}^{w'}$ accepts $G_R^{(q',y)}$ for some configuration (q', y) reachable from (q, x). That is, a configuration (q', y) for which there is $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_{\alpha_i}$, z is accepted by $\mathcal{U}_{\beta_i}$, and is y' accepted by $\mathcal{U}_{\gamma_i}$. The way $\mathcal{A}$ detects such a configuration (q, y) is the following. From the node x, the automaton $\mathcal{A}$ simulates the automaton $\mathcal{U}_{\alpha_i}$ upwards (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_{\alpha_i}$ on the word it reads as it proceeds on direction $\uparrow$ from x towards the root of the V-tree). Suppose that on its way up to the root, $\mathcal{A}$ gets to a state in F_{α_i} as it reads the node $z \in V^*$. This means that the word read so far is in α_i, and can serve as the prefix x' above. If this is indeed the case, then it is left to check that the word z is accepted by $\mathcal{U}_{\beta_i}$, and that there is a state that is obtained from z by prefixing it with a word $y' \in \gamma_i$ that is accepted by $\mathcal{S}^{s'}$. To check the first condition, $\mathcal{A}$ sends a copy in direction $\uparrow$ that simulates a run of $\mathcal{U}_{\beta_i}$, hoping to reach a state in F_{β_i} as it reaches the root (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_{\beta_i}$ on the word it reads as it proceeds from z up to the root of the V-tree). To check the second condition, $\mathcal{A}$ simulates the automaton $\mathcal{U}_{\gamma_i}$ backwards down the tree. A node $y' \cdot z \in V^*$ that $\mathcal{A}$ reads as it encounters the initial state $q^0_{\gamma_i}$ can serve as the node y we are after. The case for an atom $\Box w'$ is similar, only that here $\mathcal{A}$ needs to check whether $\mathcal{W}^s$ accepts $G_R^{(q,y)}$ for all configurations (q', y) reachable from x, and thus the choices made by $\mathcal{A}$ for guessing the partition $x' \cdot z$ of x and the prefix y' of y are dual.

In order to follow the above application of rewrite rules, the state space of $\mathcal{A}$ is $P = W \times Q \times T \times Q_\Omega \times \{\forall, \exists\}$. Thus, a state is a 5-tuple $p = \langle w, q, \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle, s, b \rangle$, where $b \in \{\forall, \exists\}$ is the simulation mode (depending on whether we are applying R to an $\Diamond$ or an $\Box$ atom), $\langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$ is the rewrite rule in T we are applying, and $s \in Q_{\alpha_i} \cup Q_{\beta_i} \cup Q_{\gamma_i}$ is the current state of the simulated automaton[6]. A state where $s = q^0_{\gamma_i}$ is an action state,where we apply R on the transitions in δ.

[6]Note that a straightforward representation of P results in $O(|Q| \cdot |T| \cdot |R| \cdot |V|)$ states. Since, however, the states of the automata for the regular expressions are disjoint, we can assume that the tuple in T that each automaton corresponds to is uniquely defined from it.

Other states are navigation states. The formal definition of the transition function of $\mathcal{A}$ follows quite straightforwardly from the definition of the state space and the explanation above.

The acceptance condition of $\mathcal{A}$ is the adjustment of F to the new state space. That is, it is obtained from F by replacing each set F_i by the set $F_i \times Q \times T \times Q^0_\gamma \times \{\forall, \exists\}$. We add $W \times Q \times T \times (Q_\Omega \setminus Q^0_\gamma) \times \{\forall\}$ as the maximal even set and $W \times Q \times T \times (Q_\Omega \setminus Q^0_\gamma) \times \{\exists\}$ as the maximal odd set. This way, in existential mode we exclude runs in which the simulation phase continues forever while allowing them in universal mode. Indeed, as we assumed that initial states have no incoming arrows, as long as $\mathcal{A}$ does not reach the initial state of $\mathcal{U}_{\gamma_i}$ it cannot visit lower sets in the acceptance condition. □

The constructions in Theorems 3.4.1 and 3.4.2 reduce the global model-checking problem to the global membership problem of a 2APT. By Theorem 3.3.1, we then have the following.

Theorem 3.4.3 *The global model-checking problem for a pushdown or a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T\rangle$ and a graph automaton $\mathcal{W} = \langle \Sigma, W, w_0, \delta, F\rangle$, can be solved in time exponential in nk, where $n = |Q| \cdot \|T\| \cdot |V|$ and k is the index of $\mathcal{W}$.*

Together with Theorem 3.2.4, we can conclude with an EXPTIME bound also for the global model-checking problem of μ-calculus formulas, matching the lower bound in [Wal96]. Note that the fact the same complexity bound holds for pushdown and prefix-recognizable rewrite systems stems from the different definition of $\|T\|$ in the two cases.

3.5 Path Automata on Trees

Path automata on trees are a hybrid of nondeterministic word automata and nondeterministic tree automata: they run on trees but have linear runs. Here we describe *two-way* nondeterministic Büchi path automata. We introduced path automata in [KPV02], where they are used to give an automata-theoretic solution to the local linear time model checking problem[7]. A *two-way nondeterministic Büchi path automaton* (2NBP, for short) on Σ-labeled Υ-trees is a 2ABT where the transition is restricted to disjunctions. Formally, $\mathcal{S} = \langle \Sigma, P, p_0, \delta, F\rangle$, where Σ, P, p_0, and F are as in an NBW, and $\delta : P \times \Sigma \to 2^{(ext(\Upsilon)\times P)}$ is the transition function. A path automaton that visits the state p and reads the node $x \in T$ chooses a pair $(d, p') \in \delta(p, \tau(x))$, and then follows direction d and moves to state p'. It follows that a *run* of a 2NBP on a labeled tree $\langle \Upsilon^*, \tau\rangle$ is a sequence of pairs $r = (x_0, p_0), (x_1, p_1), \ldots$. The run is *accepting* if it visits F infinitely often. As usual, $\mathcal{L}(\mathcal{S})$ denotes the set of trees accepted by $\mathcal{S}$. We measure the size of a 2NBP by two parameters, the number of states and the size, $|\delta| = \Sigma_{p\in P}\Sigma_{a\in\Sigma}|\delta(s, a)|$, of the transition function.

We studied in [KPV02] the emptiness and membership problems for 2NBP. Here, we consider the global membership problem of 2NBP. We show that the reduction used in [KPV02] from the membership problem of 2NBP to the emptiness problem of ABW, can be used to construct an NFW N that accepts the word $w \in \Upsilon^*$ in state $p \in P$ (i.e. the run ends in state p of $\mathcal{S}$; state p is an accepting sink of N) iff $\mathcal{S}$ accepts $\langle \Upsilon^*, \tau\rangle$ from (q, w).

[7] There is a similar type of automata called *Tree Walking Automata*. These are automata that read finite trees and expect the nodes of the tree to be labeled by the direction and by the set of successors of the node. Tree walking automata are used in XML queries. See [EHvB99, Nev02].

Theorem 3.5.1 *Consider a 2NBP $\mathcal{S} = \langle \Sigma, P, p_0, \delta, F \rangle$ and a regular tree $\langle \Upsilon^*, \tau \rangle$. We can construct an NFW $N = \langle \Upsilon, Q' \cup P, q_0, \Delta, P \rangle$ that accepts the word w in a state $p \in P$ iff $\mathcal{S}$ accepts T from (p, w). We construct N in time $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$ and space $O(|P|^2 \cdot \|\tau\|)$.*

The first thing that we do is slightly modify the 2NBP. We add an 'idle' state, in which the automaton starts its run from the root. In this idle state, the automaton navigates to some arbitrary node of the tree. Then, the automaton transitions to an arbitrary state and starts a 'normal' run. The 'idle' state masks the fact that we would like to identify all the pairs (q, w) from which the tree is accepted. Thus, the new automaton $\mathcal{S}'$ navigates to the node w in the idle state and then transitions into state q.

We showed in [KPV02] how to construct an ABW A that is not empty iff $\mathcal{S}'$ accepts the tree T. In the proof, we translate an accepting run of A on a^ω into an accepting run of $\mathcal{S}'$ on T and vice versa. Thus, there is a 1-1 and onto correspondence between runs of A on a^ω and runs of $\mathcal{S}'$ on T. We extract from the emptiness information on A the pairs (q, w) such that $\mathcal{S}^q$ accepts the tree from node w. The full proof of Theorem 3.5.1, which is rather involved, is in Appendix 3.A.

3.6 Global Linear Time Model Checking

In this section we solve the global model-checking for linear time specifications. As branching time model-checking is exponential in the system and linear time model-checking is polynomial in the system, we do not want to simply reduce linear time model-checking to branching time model-checking. We have to develop methods specifically for linear time. We solve the global model-checking problem by a reduction to the global membership problem of 2NBP. We start by demonstrating our technique on global model-checking for pushdown systems. Then we show how to extend it to prefix-recognizable systems. Again, the main difference from the construction in [KPV02] is the usage of the global-membership problem of 2NBP.

As in the previous section, the 2NBP reads the full infinite V-tree. It uses its location as the store and memorizes as part of its state the state of the rewrite system. As before, for pushdown systems it is sufficient to label a node in the tree by its direction. For prefix-recognizable systems the label is more complex and reflects the membership of x in the regular expressions that are used in the transition rules.

3.6.1 Pushdown Systems

We use again the tree $\langle V^*, \tau_V \rangle$. We construct a 2NBP $\mathcal{S}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{S}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{S}$ consults its current state and the label of the node it reads.

Theorem 3.6.1 *Given a pushdown system $R = \langle \Sigma, V, Q, L, T \rangle$ and an NBW $N = \langle \Sigma, W, w_0, \eta, F \rangle$, we can construct a 2NBP $\mathcal{S}$ on V-trees and a function f that associates states of $\mathcal{S}$ with states of R such that $\mathcal{S}$ accepts $\langle V^*, \tau_V \rangle$ from (s, x) iff $G_R^{(f(s),x)} \models N$. The automaton $\mathcal{S}$ has $O(|Q| \cdot \|T\| \cdot |N|)$ states and the size of its transition function is $O(\|T\| \cdot |N|)$.*

Proof: We define $\mathcal{S} = \langle V, P, p_0, \delta, F' \rangle$, where

- $P = W \times Q \times tails(T)$. Intuitively, when $\mathcal{S}$ visits a node $x \in V^*$ in state $\langle w, q, y\rangle$, it checks that R with initial configuration $(q, y \cdot x)$ is accepted by N^w. In particular, when $y = \varepsilon$, then R with initial configuration (q, x) needs to be accepted by N^w. As before, states of the form $\langle w, q, \varepsilon\rangle$ are *action states* where $\mathcal{S}$ imposes new requirement on $\langle V^*, \tau_V\rangle$. States of the form $\langle w, q, y\rangle$, for $y \in V^+$, are *navigation states.*

- The transition function δ is defined for every state in $\langle w, q, x\rangle \in W \times Q \times tails(T)$ and letter $A \in V$ as follows.

  - $\delta(\langle w, q, \epsilon\rangle, A) = \{(\langle w', q', y\rangle, \uparrow) \ : \ w' \in \eta(w, L(q, A))$ and $\langle q, A, y, q'\rangle \in T\}$.
  - $\delta(\langle w, q, B \cdot y\rangle, A) = \{(\langle w, q, y\rangle, B)\}$.

  Thus, in action states, $\mathcal{S}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to η. In navigation states, $\mathcal{S}$ needs to go downwards $B \cdot y$, so it continues in direction B.

- $F' = \{\langle w, q, \epsilon\rangle \ : \ w \in F$ and $\ q \in Q\}$. Note that only action states can be accepting states of $\mathcal{S}$.

The function f associates with state (w_0, q, ϵ) of $\mathcal{S}$ the state q of R. For other states f is undefined.

Assume first that $\mathcal{S}$ accepts $\langle V^*, \tau_V\rangle$ when starting its run in state (w_0, q, ϵ) from node x. Then, there exists an accepting run $r = ((w_0, q, \epsilon), x), ((w_1, q_1, \alpha_1), x_1), \ldots$ of $\mathcal{S}$ on $\langle V^*, \tau_V\rangle$. Extract from r the subsequence $((w_0, q, \epsilon), x), ((w_{i_1}, q_{i_1}, \epsilon), x_{i_1}), \ldots$ of action states. As the run is accepting and only action states are accepting states we know that this subsequence is infinite. By the definition of δ, the sequence $(q_{i_1}, x_{i_1}), (q_{i_2}, x_{i_2}), \ldots$ corresponds to an infinite path in the graph G_R. Also, by the definition of F', the run $w_0, w_{i_1}, w_{i_2}, \ldots$ is an accepting run of N on the trace of this path. Hence, G_R contains an (x, q)-trace that is accepted by N, thus $(x, q) \models N$.

Assume now that $(q, x) \models N$. Then, there exists a path $(q, x), (q_1, x_1), \ldots$ in G_R whose trace does not satisfy φ. There exists an accepting run $w_0, w_1, \ldots$ of $\mathcal{M}_{\neg\varphi}$ on this trace. The combination of the two sequence serves as the subsequence of the action states in an accepting run of $\mathcal{S}$. It is not hard to extend this subsequence to an accepting run of $\mathcal{S}$ on $\langle V^*, \tau_V\rangle$ from $((w_0, q, \epsilon), x)$. □

3.6.2 Prefix-Recognizable Systems

We now turn to consider prefix-recognizable systems. Again a configuration of a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T\rangle$ consists of a state in Q and a word in V^*. So, the store content is still a node in the tree V^*. However, in order to apply a rewrite rule it is not enough to know the direction of the node. Recall that in order to represent the configuration $(q, x) \in Q \times V^*$, our 2NBP memorizes the state q as part of its state space and it reads the node $x \in V^*$. In order to apply the rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q'\rangle$, the 2NBP has to go up the tree along a word $y \in \alpha_i$. Then, if $x = y \cdot z$, it has to check that $z \in \beta_i$, and finally guess a word $y' \in \gamma_i$ and go downwards y' to $y' \cdot z$. Finding a prefix y of x such that $y \in \alpha_i$, and a new word $y' \in \gamma_i$ is done as in the case of branching time by emulating the automata $\mathcal{U}_{\alpha_i}$ and $\mathcal{U}_{\gamma_i}$. How can the 2NBP know that $z \in \beta_i$? Instead of labeling each node $x \in V^*$ only by its direction, we can label it also by the regular expressions β for which $x \in \beta$. Thus, when the 2NBP runs $\mathcal{U}_{\alpha_i}$ up the tree, it can tell, in

every node it visits, whether z is a member of β_i or not. If $z \in \beta_i$, the 2NBP may guess that time has come to guess a word in γ_i and run $\mathcal{U}_{\gamma_i}$ down the guessed word.

Thus, in the case of prefix-recognizable systems, the nodes of the tree whose membership is checked are labeled by both their directions and information about the regular expressions β. Let $\{\beta_1, \ldots, \beta_n\}$ be the set of regular expressions β_i such that there is a rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \in T$. Let $\mathcal{D}_{\beta_i} = \langle V, D_{\beta_i}, q^0_{\beta_i}, \eta_{\beta_i}, F_{\beta_i} \rangle$ be the deterministic automaton for the language of β_i^R (where L^R is the reversed language of L). For a word $x \in V^*$, we denote by $\eta_{\beta_i}(x)$ the unique state that $\mathcal{D}_{\beta_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{\beta_i}$. For a letter $\sigma \in \Sigma$, let $\sigma[i]$, for $i \in \{0, \ldots n\}$, denote the i-th element in σ (that is, $\sigma[0] \in V$ and $\sigma[i] \in D_{\beta_i}$ for $i > 0$). Let $\langle V^*, \tau_\beta \rangle$ denote the Σ-labeled V-tree such that $\tau_\beta(\epsilon) = \langle \perp, q^0_{\beta_1}, \ldots, q^0_{\beta_n} \rangle$, and for every node $A \cdot x \in V^+$, we have $\tau_\beta(A \cdot x) = \langle A, \eta_{\beta_1}(A \cdot x), \ldots, \eta_{\beta_n}(A \cdot x) \rangle$. Thus, every node x is labeled by $dir(x)$ and the vector of states that each of the deterministic automata reach after reading x^R. Note that $\tau_\beta(x)[i] \in F_{\beta_i}$ iff $x^R \in \beta_i^R$ i.e. $x \in \beta_i$. Note also that $\langle V^*, \tau_\beta \rangle$ is a regular tree whose size is exponential in the sum of the lengths of the regular expressions $\beta_1, \ldots, \beta_n$.

Theorem 3.6.2 *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T \rangle$ and an NBW $N = \langle \Sigma, W, w_0, \eta, F \rangle$, we can construct a 2NBP $\mathcal{S}$ on V-trees and a function f that associates states of $\mathcal{S}$ with states of R such that $\mathcal{S}$ accepts $\langle V^*, \tau_\beta \rangle$ from (s, x) iff $G_R^{(f(s),x)} \models N$. The automaton $\mathcal{S}$ has $O(|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot |T| \cdot |N|)$ states and the size of its transition function is $O(\|T\| \cdot |N|)$.*

The proof resembles the proof for pushdown systems. This time, the application of a rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ involves an emulation of the automata $\mathcal{U}_{\alpha_i}$ (upwards) and $\mathcal{U}_{\gamma_i}$ (downwards). Accordingly, one of the components of the states of the 2NBP is a state of either $\mathcal{U}_{\alpha_i}$ or $\mathcal{U}_{\gamma_i}$. Action states are states in which this component is the initial state of $\mathcal{U}_{\gamma_i}$. From action states, the 2NBP chooses a new rewrite rule $t_{i'} = \langle q', \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q'' \rangle$, and it applies it as follows. First, it enters the initial state of $\mathcal{U}_{\alpha_{i'}}$, and runs $\mathcal{U}_{\alpha_{i'}}$ up the tree until it reaches a final state. It then verifies that the current node is in the language of $\beta_{i'}$, in which case it moves to a final state of $\mathcal{U}_{\gamma_{i'}}$ and runs it backward down the tree until it reaches a new action state.

Proof: We define $\mathcal{S} = \langle \Sigma, P, p_0, \delta, F' \rangle$ as follows.

- $\Sigma = V \times \Pi_{i=1}^n D_{\beta_i}$.
- $P = \{\langle w, q, s, t_i \rangle \mid w \in W,\ q \in Q,\ t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T,\ \text{ and } s \in Q_{\alpha_i} \cup Q_{\gamma_i}\}$

  Thus, S holds in its state a state of N, a state in Q, the current state in Q_α or Q_γ, and the current rewrite rule being applied. A state $\langle w, q, s, \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \rangle$ is an action state if s is the initial state of $\mathcal{U}_{\gamma_i}$, that is $s = q^0_{\gamma_i}$. In action states, $\mathcal{S}$ chooses a new rewrite rule $t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle$. Then $\mathcal{S}$ updates the N component according to the current location in the tree and moves to the state $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$. Other states are navigation states. If $s \in Q_{\gamma_i}$ is a state in $\mathcal{U}_{\gamma_i}$ (that is not initial), then $\mathcal{S}$ chooses a direction in the tree, a predecessor of the state in Q_{γ_i} reading the chosen direction, and moves in the chosen direction. If $s \in Q_{\alpha_i}$ is a state of $\mathcal{U}_{\alpha_i}$ then $\mathcal{S}$ moves up the tree (towards the root) while updating the state of $\mathcal{U}_{\alpha_i}$. If $s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ and $\tau(x)[i] \in F_{\beta_i}$ marks the current node x as a member of the language of β_i then $\mathcal{S}$ moves to an accepting state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$ (recall that initial states and accepting states have no incoming / outgoing edges respectively).

- The transition function δ is defined for every state in P and letter in $\Sigma = V \times \Pi_{i=1}^{n} D_{\beta_i}$ as follows.

$$\delta(\langle w,q,s,t_i\rangle,\sigma) = \begin{cases} \left\{ (\langle w,q,s',t_i\rangle,\uparrow) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle \\ s' \in \eta_{\alpha_i}(s,\sigma[0]) \end{array} \right\} \cup \\ \left\{ (\langle w,q,s',t_i\rangle,\epsilon) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, \\ s \in F_{\alpha_i},\ s' \in F_{\gamma_i}, \\ \text{and } \sigma[i] \in F_{\beta_i} \end{array} \right\} & s \in Q_\alpha \\ \left\{ (\langle w,q,s',t_i\rangle,B) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle \\ s \in \eta_{\gamma_i}(s',B) \text{ and } B \in V \end{array} \right\} \cup \\ \left\{ (\langle w',q'',s',t_{i'}\rangle,\epsilon) \;\middle|\; \begin{array}{l} t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, \\ t_{i'} = \langle q,\alpha_{i'},\beta_{i'},\gamma_{i'},q''\rangle, \\ w' \in \eta(w,L(q,\sigma[0])), \\ s = q^0_{\gamma_i} \text{ and } s' = q^0_{\alpha_{i'}} \end{array} \right\} & s \in Q_\gamma \end{cases}$$

  Thus, when $s \in Q_\alpha$ the 2NBP $\mathcal{S}$ either chooses a successor s' of s and goes up the tree or in case s is an accepting state of $\mathcal{U}_{\alpha_i}$ and $\sigma[i] \in F_{\beta_i}$ then $\mathcal{S}$ chooses an accepting state of $\mathcal{U}_{\gamma_i}$.

  When $s \in Q_\gamma$ the 2NBP $\mathcal{S}$ either guesses a direction B and chooses a B-predecessor s' of s or in case $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{S}$ updates the state of N, chooses a new rewrite rule $t_{i'} = \langle q,\alpha_{i'},\beta_{i'},\gamma_{i'},q''\rangle$ and moves to the initial state $q^0_{\alpha_{i'}}$ of $\mathcal{U}_{\alpha_{i'}}$.

- $F' = \{\langle w,q,s,t_i\rangle \mid w \in F,\ q \in Q,\ t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, \text{ and } s = q^0_{\gamma_i}\}$

  Only action states may be accepting. As initial states (of $\mathcal{U}_{\gamma_i}$) have no incoming edges, in an accepting run, no navigation stage can last indefinitely.

The function f associates with state $(w_0,q,q^0_{\gamma_i},\langle q',\alpha_i,\beta_i,\gamma_i,q\rangle)$ the state q of R. For other states, f is undefined.

As before we can show that a (s,x) trace that satisfies N and the rewrite rules used to create this trace can be used to produce a run of $\mathcal{S}$ on $\langle V^*,\tau_\beta\rangle$ starting from node x in state (w_0,q,s,t_i) where $t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle$ and $s = q^0_{\gamma_i}$.

Similarly, an accepting run of $\mathcal{S}$ on $\langle V^*,\tau_\beta\rangle$ starting from node x in state (w_0,q,s,t_i) where $t_i = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle$ and $s = q^0_{\gamma_i}$ is used to find a (q,x)-trace in G_R that is accepted by N. □

Notice that there is some redundancy in the states of $\mathcal{S}$. If we assume that a transition $\langle q,\alpha_i,\beta_i,\gamma_i,q'\rangle$ is recognized by the states in $Q_{\alpha_i} \cup Q_{\gamma_i}$, then we can remove the T component from $\mathcal{P}$. Combining Theorems 3.5.1, 3.6.1 and 3.6.2, we get the following.

Theorem 3.6.3 *The global model-checking problem for a rewrite system R and NBW N is solvable*

- *in time $O((\|T\| \cdot |N|)^3)$ and space $O((\|T\| \cdot |N|)^2)$ when R is a pushdown system.*
- *in time $(\|T\| \cdot |N|)^3 \cdot 2^{O(|Q_\beta|)}$ and space $(|T| \cdot |N|)^2 \cdot 2^{O(|Q_\beta|)}$ when R is a prefix-recognizable system.*

Our complexity coincides with the one in [EHRS00], for pushdown systems, and with the result of combining [EKS01] and [KPV02], for prefix-recognizable systems.

Bibliography

[ATM03] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive game graphs. In *Computer-Aided Verification, Proc. 15th International Conference*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2003.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BCMS00] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. Unpublished manuscript, 2000.

[BE96] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1996.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, July 1997. Springer-Verlag.

[BQ96] O. Burkart and Y.-M. Quemener. Model checking of infinite graphs defined by graph grammars. In *Proc. 1st International workshop on verification of infinite states systems*, volume 6 of *ENTCS*, page 15. Elsevier, 1996.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130, Stanford, CA, USA, August 2000. Springer-Verlag.

[BR01] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. 3rd Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 1992.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic J. Comut.*, 2:89–125, 1995.

[BS99] O. Burkart and B. Steffen. Model checking the full modal μ-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Bur97a] O. Burkart. Automatic verification of sequential infinite-state processes. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Lecture Notes in Computer Science*, volume 1354. Springer-Verlag, 1997.

[Bur97b] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. In F. Moller, editor, *Proc. 2nd International workshop on verification of infinite states systems*, 1997.

[Cac02a] T. Cachat. Two-way tree automata solving pushdown games. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, chapter 17, pages 303–317. Springer-Verlag, November 2002.

[Cac02b] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. In *4th International Workshop on Verification of Infinite-State Systems*, *Electronic Notes in Theoretical Computer Science* 68(6), Brno, Czech Republic, August 2002.

[Cac03] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *Proc. 30th International Colloqium on Automata, Languages, and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569, Eindhoven, The Netherlands, June 2003. Springer-Verlag.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 1996.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CKKV01] H. Chockler, O. Kupferman, R.P. Kurshan, and M.Y. Vardi. A practical approach to coverage in model checking. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 66–78. Springer-Verlag, 2001.

[CKV01] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. In *7th International Conference on Tools and algorithms for the construction and analysis of systems*, number 2031 in Lecture Notes in Computer Science, pages 528 – 542. Springer-Verlag, 2001.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[CW02] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proc. 9th ACM conference on Computer and Communications Security*, pages 235–244, Washington, DC, USA, 2002. ACM.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247, Chicago, IL, July 2000. Springer-Verlag.

[EHvB99] J. Engelfriet, H.J. Hoggeboom, and J.-P van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, San Juan, October 1991.

[EJS93] E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Comptuer Science*, pages 385–396, Elounda, Crete, June 1993. Springer-Verlag.

[EKS01] J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 316–339, Sendai, Japan, October 2001. Springer-Verlag.

[Eme97] E.A. Emerson. Model checking and the μ-calculus. In N. Immerman and Ph.G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society, 1997.

[ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336, Paris, France, July 2001. Springer-Verlag.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown automata. In F. Moller, editor, *Proc. 2nd International Workshop on Verification of Infinite States Systems*, 1997.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *Proc. 20th International Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 552–562. Springer-Verlag, 1995.

[KNU03] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, Grenoble, France, April 2003. Springer-Verlag.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KPV02] O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 2002.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[KV00] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2000.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[Nev02] F. Neven. Automata, logic, and XML. In *16th International Workshop on Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26, Edinburgh, Scotland, September 2002. Springer-Verlag.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97, Genova, Italy, April 2001. Springer-Verlag.

[Rab72] M.O. Rabin. Automata on infinite objects and Church's problem. *Amer. Mathematical Society*, 1972.

[Sch02] S. Schwoon. *Model-checking pushdown systems*. PhD thesis, Technische Universität München, 2002.

[Sha00] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *Proc. 11th International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16, University Park, PA, USA, August 2000. Springer-Verlag.

[SW91] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th International Coll. on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer-Verlag, Berlin, July 1998.

[VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer-Verlag, 1996.

[Wil99] T. Wilke. CTL$^+$ is exponentially more succinct than CTL. In C. Pandu Ragan, V. Raman, and R. Ramanujam, editors, *Proc. 19th conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1999.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 185–194, Tucson, 1983.

3.A Global Membership of 2NBP

3.A.1 Definition of Alternating Automata on Infinite Words

An *alternating Büchi automaton on words* (*ABW* for short) is $A = \langle \Sigma, Q, q_0, \eta, F \rangle$ where Σ, Q, q_0, and F are as in NBW and $\eta : Q \times \Sigma \to B^+(\{0,1\} \times Q)$ is the transition function. A *run* of A on an infinite word $w = w_0 w_1 \ldots$ is a labeled $\mathbb{N}$ -tree (T, r) where $r : T \to \mathbb{N} \times Q$. A node x labeled by (i, q) describes a copy of the automaton in state q reading letter w_i. The labels of a node and its successors have to satisfy the transition function η. Formally, $\epsilon \in T$ and $r(\epsilon) = (0, q_0)$ and for all nodes x with $r(x) = (i, q)$ and $\eta(q, w_i) = \theta$ there is a (possibly empty) set $\{(\Delta_1, q_1), \ldots, (\Delta_n, q_n)\} \models \theta$ such that $\{x \cdot 1, \ldots, x \cdot n\} \subseteq T$ and for every $1 \leq c \leq n$ we have , $r(x \cdot c) = (i + \Delta_c, q_c)$. Thus, a 0-transition leaves the automaton reading the same letter. Note that for 2NBP we call transitions that leave the automaton in the same location ϵ-transitions and for ABW we call them 0-transitions.

A run of an ABW is *accepting* if every infinite path visits the accepting set infinitely often. As before, a word w is *accepted* by A if A has an accepting run on the word. We similarly define the language $L(A)$ of A.

Again, the size of the automaton is determined by the number of its states and the size of its transition function. The size of the transition function is $|\eta| = \Sigma_{q \in Q} \Sigma_{a \in \Sigma} |\eta(q, a)|$ where, for a formula in $B^+(\{0,1\} \times Q)$ we define $|(\Delta, q)| = |\mathbf{true}| = |\mathbf{false}| = 1$ and $|\theta_1 \vee \theta_2| = |\theta_1 \wedge \theta_2| = |\theta_1| + |\theta_2| + 1$.

Theorem 3.A.1 [VW86] *Given an ABW over 1-letter alphabet $A = \langle \{a\}, Q, q_0, \eta, F \rangle$ we can check whether $L(A)$ is empty in time $O(|\eta|)$ and space $O(|Q|)$.*

The emptiness algorithm can also produce a table $T : Q \to \{0,1\}$ such that $T(q) = 1$ iff $L(A^q) \neq \emptyset$. A simple extension of the algorithm can produce for a state q such that $L(A^q) \neq \emptyset$ an accepting (ultimately periodic) run of A^q on a^ω.

3.A.2 Construction of the NFW

Theorem 3.A.1 *Consider a 2NBP $\mathcal{S} = \langle \Sigma, P, p, \delta, F \rangle$ and a regular tree $T = \langle \Upsilon^*, \tau \rangle$. We can construct an NFW $N = \langle \Upsilon, Q' \cup P, q_0, \Delta, P \rangle$ that accepts the word w in a state $p \in P$ iff $\mathcal{S}$ accepts T from (p, w). We construct N in time $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$ and space $O(|P|^2 \cdot \|\tau\|)$.*

Proof: Consider the 2NBP $\mathcal{S}' = \langle \Sigma, P', p_0, \delta', F \rangle$ where $P' = P \cup \{p_0\}$ and $p_0 \notin P$ is a new state, for every $p \in P$ and $\sigma \in \Sigma$ we have $\delta'(p, \sigma) = \delta(p, \sigma)$, and for every $\sigma \in \Sigma$ we have $\delta'(p_0, \sigma) = \bigvee_{v \in \Upsilon}(p_0, v) \vee \bigvee_{p \in P}(\varepsilon, p)$. Thus, $\mathcal{S}'$ starts reading $\langle \Upsilon^*, \tau \rangle$ from the root in state p_0, the transition of p_0 includes either transitions down the tree that remain in state p_0 or transitions into one of the other states of $\mathcal{S}$. Thus, every accepting run of $\mathcal{S}'$ starts with a sequence $(p_0, w_0), (p_0, w_1), \ldots, (p_0, w_n), (p, w_n), \ldots$. Such a run is a witness to the fact that $\mathcal{S}$ accepts $\langle \Upsilon^*, \tau \rangle$ from (p, w_n). We would like to recognize all words $w \in \Upsilon^*$ and states $p' \in P$ for which there exist runs as above with $p = p'$ and $w_n = w$.

Consider the regular tree $\langle \Upsilon^*, \tau \rangle$. Let $\mathcal{D}_\tau$ be the transducer that generates the labels of τ where $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D_\tau, d^0_\tau, \rho_\tau, L_\tau \rangle$. For a word $w \in \Upsilon^*$ we denote by $\rho_\tau(w)$ the unique state that $\mathcal{D}_\tau$ gets to after reading w. In [KPV02] we construct the ABW $\mathcal{A} = \langle \{a\}, Q, q_0, \eta, F' \rangle$ as follows.

- $Q = (P' \cup (P' \times P')) \times D_\tau \times \{\bot, \top\}$.
- $q_0 = \langle p_0, d^0_\tau, \bot \rangle$.
- $F' = (F \times D_\tau \times \{\bot\}) \cup (P' \times D_\tau \times \{\top\})$.

In order to define the transition function we have the following definitions. Two functions $f_\alpha : P' \times P' \to \{\bot, \top\}$ where $\alpha \in \{\bot, \top\}$, and for every state $p \in P'$ and alphabet letter $\sigma \in \Sigma$ the set C^σ_p is the set of states from which p is reachable by a sequence of ϵ-transitions reading letter σ and one final $\uparrow$-transition reading σ. Formally

$$f_\bot(p, q) = \bot$$

$$f_\top(p, q) = \begin{cases} \bot & \text{if } p \in F \text{ or } q \in F \\ \top & \text{otherwise} \end{cases}$$

$$C^\sigma_p = \left\{ p' \;\middle|\; \begin{array}{l} \exists s_0, s_1, \ldots, s_n \in (P')^+ \text{ such that} \\ s_0 = p',\ s_n = p, \\ \forall 0 < i < n,\ \langle \epsilon, s_i \rangle \in \delta'(s_{i-1}, \sigma), \text{ and} \\ \langle \uparrow, s_n \rangle \in \delta(s_{n-1}, \sigma) \end{array} \right\}$$

Now η is defined for every state in Q as follows.

$$\eta(p, d, \alpha) = \bigvee \begin{array}{l} \bigvee_{p' \in P'} \bigvee_{\beta \in \{\bot, \top\}} (\langle p, p', d, \beta \rangle, 0) \wedge (\langle p', d, \beta \rangle, 0) \\ \bigvee_{v \in \Upsilon} \bigvee_{\langle v, p' \rangle \in \delta'(p, L_\tau(d))} (\langle p', \rho_\tau(d, v), \bot \rangle, 1) \\ \bigvee_{\langle \epsilon, p' \rangle \in \delta'(p, L_\tau(d))} (\langle p', d, \bot \rangle, 0) \end{array}$$

$$\eta(p_1, p_2, d, \alpha) = \bigvee \begin{array}{l} \bigvee_{\langle \epsilon, p' \rangle \in \delta'(p_1, L_\tau(d))} (\langle p', p_2, d, f_\alpha(p', p_2) \rangle, 0) \\ \bigvee_{p' \in P'} \bigvee_{\beta_1 + \beta_2 = \alpha} \left(\begin{array}{l} (\langle p_1, p', d, f_{\beta_1}(p_1, p') \rangle, 0) \wedge \\ (\langle p', p_2, d, f_{\beta_2}(p', p_2) \rangle, 0) \end{array} \right) \\ \bigvee_{v \in \Upsilon, \langle v, p' \rangle \in \delta'(p_1, L_\tau(d))} \bigvee_{p'' \in C^{L_\tau(d)}_{p_2}} (\langle p', p'', \rho_\tau(d, v), f_\alpha(p', p'') \rangle, 1) \end{array}$$

Finally, we replace every state of the form $\{\langle p, p, d, \alpha \rangle \mid \text{either } p \in F \text{ or } \alpha = \bot\}$ by **true**.

The following claim establishes the connection between $\mathcal{A}$ and $\mathcal{S}'$.

Claim 3.A.2 [KPV02] $\mathcal{L}(\mathcal{A}) \neq \emptyset$ *iff* $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{S})$

The proof in [KPV02] translates an accepting run of $\mathcal{S}'$ on $\langle \Upsilon^*, \tau \rangle$ into an accepting run tree of $\mathcal{A}$ on a^ω and vice versa. It follows from the proof, that whenever the language of a state (p, d, α) is not empty, then there exists an accepting run of $\mathcal{S}'$ on the regular tree $\langle \Upsilon^*, \tau_d \rangle$ where τ_d is the labeling induced by the transducer $\mathcal{D}^d$. Similarly, whenever the language of a state (p_1, p_2, d, α) is not empty, then there exists a partial run of $\mathcal{S}'$ that starts and ends in the root of $\langle \Upsilon^*, \tau_d \rangle$. Furthermore, if $\alpha = \top$ then this partial run contains a state in F.

As shown in [KPV02] the number of states of $\mathcal{A}$ is $O(|P|^2 \cdot \|\tau\|)$ and the size of its transition is $O(|\delta| \cdot |P|^2 \cdot \|\tau\|)$. It is also shown there that because of the special structure of $\mathcal{A}$ its emptiness can be computed in space $O(|P|^2 \cdot \|\tau\|)$ and in time $O(|\delta| \cdot |P|^2 \cdot \|\tau\|)$. As previously explained, from the emptiness algorithm we can get a table $T : Q \to \{0, 1\}$ such that $T(q) = 1$ iff $L(\mathcal{A}^q) \neq \emptyset$. Furthermore, we can extract from the algorithm an accepting run of $\mathcal{A}^q$ on a^ω. It follows that in case $(p, d, \alpha) \in P \times D_\tau \times \{\bot, \top\}$ the run is infinite and the algorithm in [KPV02] can be used to extract from it an accepting run of P on the regular tree $\langle \Upsilon^*, \tau_d \rangle$. If $(p, p', d, \alpha) \in P \times P \times D_\tau \times \{\bot, \top\}$ the run is finite and the algorithm in [KPV02] can be used to extract from it a run of P on the regular tree $\langle \Upsilon^*, \tau_d \rangle$ that starts in state p and ends in state p' both reading the root of Υ^*.

We are now ready to construct the NFW N. Let $N = \langle \Upsilon, Q' \cup P, q_0, \Delta, P \rangle$ where $Q' = (\{p_0\} \cup (\{p_0\} \times P)) \times D_\tau \times \{\bot, \top\}$ and P is the set of states of $\mathcal{S}$ (that serves also as the set of accepting states), $q_0 = (p_0, d^0_\tau, \bot)$ is the initial state of $\mathcal{A}$, and Δ is defined as follows.

Consider a state $(p_0, d, \alpha) \in Q'$, its transition in $\mathcal{A}$ is

$$\eta(p_0, d, \alpha) = \bigvee \begin{array}{l} \bigvee_{p \in P} \bigvee_{\beta \in \{\bot, \top\}} (\langle p_0, p, d, \beta \rangle, 0) \wedge (\langle p, d, \beta \rangle, 0) \\ \bigvee_{\upsilon \in \Upsilon} (\langle p_0, \rho_\tau(d, \upsilon), \bot \rangle, 1) \\ \bigvee_{p \in P} (\langle p, d, \bot \rangle, 0) \end{array}$$

For every $\upsilon \in \Upsilon$ such that the language of$(p_0, \rho_\tau(d, \upsilon), \bot)$ is not empty, we add $(p_0, \rho_\tau(d,\ \upsilon), \bot)$ to $\Delta((p_0, d, \alpha), \upsilon)$. For every state p such that the language of (p_0, p, d, β) is not empty and the language of (p, d, β) is not empty, we add (p_0, p, d, β) to $\Delta((p_0, d, \alpha), \varepsilon)$. For every state $p \in P$ such that the language of $(p, d, \bot)$ is not empty, we add (the accepting state) p to $\Delta((p_0, d, \alpha), \varepsilon)$.

Consider a state $(p_0, p, d, \alpha) \in Q'$, its transition in $\mathcal{A}$ is

$$\eta(p_0, p, d, \alpha) = \bigvee \begin{array}{l} \bigvee_{p' \in P} (\langle p', p, d, f_\alpha(p', p) \rangle, 0) \\ \bigvee_{p' \in P} \bigvee_{\beta_1 + \beta_2 = \alpha} \begin{pmatrix} (\langle p_0, p', d, f_{\beta_1}(p_0, p') \rangle, 0) \wedge \\ (\langle p', p, d, f_{\beta_2}(p', p) \rangle, 0) \end{pmatrix} \\ \bigvee_{\upsilon \in \Upsilon} \bigvee_{p' \in C_p^{L_\tau(d)}} (\langle p_0, p', \rho_\tau(d, \upsilon), f_\alpha(p_0, p') \rangle, 1) \end{array}$$

For every $\upsilon \in \Upsilon$ and for every $p' \in C_p^{L_\tau(d)}$ such that the language of $(p_0, p', \rho_\tau(d, \upsilon), f_\alpha(p_0\ , p'))$ is not empty, we add $(p_0, p', \rho_\tau(d, \upsilon), f_\alpha(p_0,\ p'))$ to $\Delta((p_0, p', d, \alpha), \upsilon)$. For every state p' such that the language of $(p', p, d, f_\alpha(p', p))$ is not empty, we add p' to $\Delta((p_0, p, d, \alpha), \varepsilon)$. For every state p' such that the language of (p_0, p', d, β_1) is not empty and the language of (p', p, d, β_2) is not empty, we add (p_0, p', d, β_1) to $\Delta((p_0, p, d, \alpha), \varepsilon)$.

This completes the definition of the automaton. We have to show that for every word $w \in \Upsilon^*$ accepted by N in state $p \in P$ we have that $\langle \Upsilon^*, \tau \rangle$ is accepted by $\mathcal{S}$ from (s, w).

Lemma 3.A.3 *A word $w \in \Upsilon^*$ is accepted by N in a state $p \in P$ iff $\mathcal{S}$ accepts $\langle \Upsilon^*, \tau \rangle$ from (p, w).*

Proof: Consider some run $r = n_0, n_1, \ldots, n_l$ of N. Denote by $word(r, i)$ the sequence $v_1 \cdots v_m$ of letters read by N in the run $n_0, \ldots n_i$.

Suppose that N accepts w. There exists an accepting run r of N on w. The run r has the following form $r = (p_0, d_0, \alpha_0), \ldots, (p_0, d_n, \alpha_n), (p_0, p'_1, d'_1, \alpha'_1), \ldots (p_0, p'_k, d'_k, \alpha'_k), s$. It is simple to see that $w = word(r, n + k)$. We construct an accepting run of $\mathcal{S}$ on $\langle \Upsilon^*, \tau \rangle$ starting from (w, s). Consider the state $(p_0, p'_1, d'_1, \alpha'_1)$. From the definition of N it follows that the language of (p'_1, d'_1, α'_1) is not empty. Hence, there exists an accepting run tree of $\mathcal{S}$ starting from p' that accepts $\langle \Upsilon^*, \tau_{d'_1} \rangle$. We change this accepting run into an accepting run of $\mathcal{S}$ that starts from $word(r, n + 1)$. This serves as the suffix of our run. Consider the transition from $(p_0, p'_i, d'_i, \alpha'_i)$ to $(p_0, p'_{i+1}, d'_{i+1}, \alpha'_{i+1})$. According to the definition of N it results from one of the following:

- The disjunct $(p_0, p'_{i+1}, d'_{i+1}, \alpha'_{i+1}) \wedge (p'_{i+1}, d'_{i+1}, p'_i, \beta)$ where $d_{i+1} = d_i$ and it is an ϵ transition.
- The disjunct $(p_0, p'_{i+1}, d'_{i+1}, \alpha'_{i+1})$ where $d'_{i+1} = \rho_\tau(d'_i, v)$, $word(r, n+i+1) = word(r, n+i) \cdot v$, $p'_{i+1} \in C_{p'_i}^{L_\tau(d)}$ and the transition reads the letter v.

In the first case, there exists a run segment that connects p'_{i+1} to p'_i that starts and ends in the root of $\langle \Upsilon^*, \tau_{d_i} \rangle$. We change this run to start and end in $word(r, n + i)$ and add it before the current suffix of the run of $\mathcal{S}$. In the second case, we add the state p'_{i+1} reading $word(r, n + i + 1)$ before the current suffix. By the fact that $p'_{i+1} \in C_{p'_i}^{L_\tau(d)}$ this is a valid transition of $\mathcal{S}$.

The last transition of r adds the initial state p before the current suffix and we are done.

In the other directions, suppose that $\mathcal{S}$ accepts T from (w, s). We construct an accepting run of $\mathcal{S}'$ that starts from the root of T by padding the run with a prefix of p_0 states. We translate this run of $\mathcal{S}'$ into an accepting run of $\mathcal{A}$ as in [KPV02]. The run of N follows the prefix of the run of $\mathcal{A}$ that contains p_0 and ends in s. ▟

□

Chapter 4

Pushdown Specifications

Traditionally, model checking is applied to finite-state systems and regular specifications. While researchers have successfully extended the applicability of model checking to infinite-state systems, almost all existing work still consider regular specification formalisms. There are, however, many interesting non-regular properties one would like to model check.

In this paper we study model checking of *pushdown specifications*. Our specification formalism is nondeterministic pushdown parity tree automata (PD-NPT). We show that the model-checking problem for regular systems and PD-NPT specifications can be solved in time exponential in the system and the specification. Our model-checking algorithm involves a new solution to the nonemptiness problem of nondeterministic pushdown tree automata, where we improve the best known upper bound from a triple-exponential to a single exponential. We also consider the model-checking problem for context-free systems and PD-NPT specifications and show that it is undecidable.

4.1 Introduction

One of the most significant developments in the area of formal design verification is the discovery of algorithmic methods for verifying on-going behaviors of reactive systems [QS81, LP85, CES86, VW86]. In *model checking*, we verify the correctness of a system with respect to a desired behavior by checking whether a mathematical model of the system satisfies a formal specification of this behavior (for a survey, see [CGP99]). Traditionally, model checking is applied to *finite-state* systems, typically modeled by labeled state-transition graphs, and to behaviors that are formally specified as *temporal-logic* formulas or *automata on infinite objects*. Symbolic methods that enable model checking of very large state spaces, and the great ease of use of fully algorithmic methods, led to industrial acceptance of model checking [BLM01, CFF+01].

In recent years, researchers have tried to extend the applicability of model checking to infinite-state systems. An active field of research is model checking of *infinite-state sequential systems*. These are systems in which each state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The origin of this research is the result of Müller and Schupp that the monadic second-order theory of *context-free graphs* is decidable [MS85]. As the complexity involved in that decidability result is nonelementary, researchers sought decidability results of elementary complexity. Various algorithms for simpler logics and more general systems have been proposed. The most powerful result so far is an exponential-time algorithm by Burkart for model

checking formulas of the μ-calculus with respect to prefix-recognizable graphs [Bur97b]. See also [BS95, Cau96, Wal96, BE96, BQ96, BEM97, Bur97a, FWW97, BS99, BCMS00, KV00] and a short summary in [Tho01].

An orthogonal line of research considers the applicability of model checking to *infinite-state specifications*. Almost all existing work on model checking considers specification formalisms that define *regular* sets of words, trees, or graphs: formulas of LTL, μ-calculus, and even monadic-second order logic can all be translated to automata [Büc62, Rab69, EJ91], and in fact many model-checking algorithms (for both finite-state and infinite-state systems) first translate the given specification into an automaton and reason about the structure of this automaton (cf., [VW86, BEM97, KV00]). Sometimes, however, the desired behavior is non regular and cannot be specified by a finite-state automaton. Consider for example the property "p is inevitable", for a proposition p. That is, in every computation of the system, p eventually holds. Clearly, this property is regular and is expressible as $\forall\Diamond p$ in both CTL [CES86] and LTL [Pnu77]. On the other hand, the property "p is uniformly inevitable", namely, there is some time i such that in every computation of the system, p holds at time i, is not expressible by a finite automaton on infinite trees [Eme87], and hence, it is non regular. As another example, consider a system that handles requests and acknowledgments, and the property "every acknowledgment is preceded by some request". Again, this property is regular and is expressible in LTL as $(\neg ack)\mathcal{W}req$. On the other hand, consider the property of "no redundant acknowledgments", namely the number of acknowledgments does not exceed the number of requests. The technique of [Eme87] can be used in order to show that the property is non regular. More examples to useful non-regular properties are given in [SCFG84], where the specification of unbounded message buffers is considered.

The need to specify non-regular behaviors led Bouajjani et al. [BER94, BEH95] to consider logics that are a combination of CTL and LTL with Presburger Arithmetic. The logics, called PCTL and PLTL, use variables that range over natural numbers. The variables are bound to the occurrences of state formulas and comparison between such variables is allowed. The non-regular properties discussed above can be specified in PCTL and PLTL. For example, we can specify uniform inevitability in PCTL as $\widetilde{\exists} i \,.\, \forall[x : \mathbf{true}](x = i \rightarrow p)$, where the $\widetilde{\exists}$ quantifier quantifies over natural numbers, the $\forall$ quantifier quantifies over computations of the system, and the combinator $[x : \mathbf{true}]$ binds the variable x to count the number of occurrences of the state formula **true**. Bouajjani et al. consider the model-checking problem for the logics PCTL and PLTL over finite-state (regular) systems and over infinite-state (non-regular) systems. The logics turned out to be too strong: the model checking of both PCTL and PLTL over finite-state systems is undecidable. They proceed to restrict the logics to fragments for which model checking of finite-state systems and context-free systems is decidable. The properties "p is uniformly inevitable" and "no redundant acknowledgments" are both expressible in the restricted (decidable) fragments of PCTL and PLTL.

Uniform inevitability is clearly expressible by a *nondeterministic pushdown tree automaton*. Pushdown tree automata are finite-state automata augmented by a pushdown store. Like a nondeterministic finite-state tree automaton, a nondeterministic pushdown tree automaton starts reading a tree from the root. At each node of the tree, the pushdown automaton consults the transition relation and splits into independent copies of itself to each of the node's successors. Each copy has an independent pushdown store that diverges from the pushdown store of the parent. We then check what happens along every branch of the run tree and determine acceptance. In order to express uniform inevitability, the automaton guesses the time i, pushes i elements into the pushdown store, and, along every computation, pops one element with every move of the system. When the

pushdown store becomes empty, the automaton requires p to hold. Similarly, in order to express "no redundant acknowledgments", a nondeterministic pushdown tree automaton can push an element into the pushdown store whenever the system sends a request, pop one element with every acknowledgment, and reject the tree when an acknowledgment is issued when the pushdown store is empty. In [PI95], Peng and Iyer study more properties that are non regular and propose to use nondeterministic pushdown tree automata as a strong specification formalism. The model studied by [PI95] is *empty store*: a run of the automaton is accepting if the automaton's pushdown store gets empty infinitely often along every branch in the run tree.

In this paper we study the model-checking problem for specifications given by nondeterministic pushdown tree automata. We consider both finite-state (regular) and infinite-state (non-regular) systems. We show that for finite-state systems, the model-checking problem is solvable in time exponential in both the system and the specification, even for nondeterministic pushdown parity tree automata – a model that is much stronger than the one studied in [PI95]. On the other hand, the model-checking problem for context-free systems is undecidable – already for a weak type of pushdown tree automata. Note that by having tree automata as our specification formalism, we follow here the branching-time paradigm, where the specification describes allowed computation trees and a system is correct if its computation tree is allowed [CES86]. In Remark 4.4.2, we discuss the undecidability of the linear-time paradigm, and the reasons that make the (seemingly more general) branching-time framework decidable.

In order to solve the model-checking problem for nondeterministic pushdown tree automata and finite-state systems, we use the automata theoretic approach to branching-time model checking [KVW00]. In [KVW00], model checking is reduced to the *emptiness* problem for nondeterministic finite tree automata, here we reduce the model-checking problem to the emptiness problem for nondeterministic pushdown tree automata. The first to show that this emptiness problem is decidable were Harel and Raz [HR94]. The automata considered by Harel and Raz use the Büchi acceptance condition, where some states are designated as accepting states and a run is accepting if it visits the accepting states infinitely often along every branch in the run tree. It is shown in [HR94] that the problem can be solved in triple-exponential time. Recall that Peng and Iyer [PI95] consider a simpler acceptance condition, where a run is accepting if the automaton's pushdown store gets empty infinitely often along every branch in the run tree. For this acceptance condition, it is shown in [PI95] that the nonemptiness problem can be solved in exponential time. Nevertheless, empty store pushdown automata are strictly weaker than nondeterministic Büchi pushdown tree automata [PI95] and the algorithm in [PI95] cannot be extended to handle the Büchi acceptance condition.

The main result of this paper is an exponential algorithm for the emptiness problem of nondeterministic *parity* pushdown tree automata. Thus, apart from improving the known triple-exponential upper bound to a single exponential, we handle a more general acceptance condition. Our algorithm is based on a reduction of the emptiness problem to the membership problem for *two-way alternating parity tree automata* with no pushdown store. We note that our technique can be applied also to specifications given by *alternating* pushdown parity tree automata. Indeed, the automata-theoretic approach to branching-time model checking involves some type of a product between the system and the specification automaton, making alternation as easy as nondeterminism [KVW00]. In Remark 4.4.3, we discuss this point further, and also show that, unlike the case of regular automata, alternating pushdown automata are strictly more expressive than nondeterministic pushdown tree automata.

Once one realizes that the difficulties in handling the pushdown store of the tree automaton

are similar to the difficulties in handling the pushdown store of infinite-state sequential systems, it is possible to solve the nonemptiness problem for pushdown automata with various methods that have been suggested for the latter. In particular, it is possible to reduce the nonemptiness problem for nondeterministic pushdown parity tree automata to the μ-calculus model-checking problem for pushdown systems [Wal01]. The solution we suggest here is the first to suggest the application of methods developed for reasoning about infinite-state sequential systems to the solution of automata-theoretic problems for pushdown automata. In particular, we believe that methods based on two-way alternating tree automata [KV00, KPV02a] are particularly appropriate for this task, as the solution stays in the clean framework of automata.

Finally, in order to show the undecidability result, we reduce the problem of deciding whether a two-counter machine accepts the empty tape to the model-checking problem of a context-free system with respect to a nondeterministic pushdown tree automaton. Intuitively, the pushdown store of the system can simulate one counter, and the pushdown store of the specification can simulate the second counter.

The study of pushdown specifications completes the picture described in the table below regarding model checking of regular and context-free systems with respect to regular and pushdown specifications. When both the system and the specification are regular, the setting is that of traditional model checking [CGP99]. When only one parameter has a pushdown store, the problem is still decidable. Yet, when both the system and the specification have a pushdown store, model checking becomes undecidable. The complexities in the table refer to the case where the specification is given by a nondeterministic or an alternating parity tree automaton of size n and index k. The size of the system is m.

	Regular Specifications	Pushdown Specifications
Regular Systems	decidable; $O((nm)^k)$ [EJS93]	decidable; $exp(mnk)$ [Theorem 4.4.1]
Pushdown Systems	decidable; $exp(mnk)$ [KV00]	undecidable [Theorem 4.5.1]

Figure 4.1: Model-checking regular and pushdown systems and specifications.

An preliminary version of this paper appeared in [KPV02b].

4.2 Definitions

4.2.1 Trees

Given a finite set Υ of directions, an Υ-*tree* is a set $T \subseteq \Upsilon^*$ such that if $v \cdot x \in T$, where $v \in \Upsilon$ and $x \in \Upsilon^*$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $v \in \Upsilon$ and $x \in T$, the node x is the *parent* of $v \cdot x$ and $v \cdot x$ is a *successor* of x. If $z = x \cdot y \in T$ then z is a descendant of y. Each node $x \neq \varepsilon$ of T has a *direction* in Υ. The direction of the root is the symbol $\perp$ (we assume that $\perp \notin \Upsilon$). The direction of a node $v \cdot x$ is v. We denote by $dir(x)$ the direction of the node x. An Υ-tree T is the *full infinite tree* if $T = \Upsilon^*$. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $v \in \Upsilon$ such that $v \cdot x \in \pi$. Note that our definitions here reverse the standard definitions (e.g., when $\Upsilon = \{0, 1\}$, the successors of the node 0 are 00 and 10, rather than 00 and 01)[1].

[1] As will get clearer in the sequel, the reason for that is that rewrite rules refer to the prefix of words.

Given two finite sets Υ and Σ, a Σ-*labeled* Υ-*tree* is a pair $\langle T, \tau \rangle$ where T is an Υ-tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, \tau \rangle$ a labeled tree. A tree is *regular* if it is the unwinding of some finite labeled graph. More formally, a *transducer* is a tuple $\mathcal{D} = \langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, where Υ is a finite set of directions, Σ is a finite alphabet, Q is a finite set of states, $\eta : Q \times \Upsilon \to Q$ is a deterministic transition function, $q_0 \in Q$ is an initial state, and $L : Q \to \Sigma$ is a labeling function. We define $\eta : \Upsilon^* \to Q$ in the standard way: $\eta(\varepsilon) = q_0$ and for $x \in \Upsilon^*$ and $\upsilon \in \Upsilon$ we have $\eta(\upsilon x) = \eta(\eta(x), \upsilon)$. Intuitively, a transducer is a labeled finite graph with a designated start node, where the edges are labeled by Υ and the nodes are labeled by Σ. A Σ-labeled Υ-tree $\langle \Upsilon^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{D} = \langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, such that for every $x \in \Upsilon^*$, we have $\tau(x) = L(\eta(x))$. We then say that the size of the regular tree $\langle \Upsilon^*, \tau \rangle$, denoted $\|\tau\|$, is $|Q|$, the number of states of $\mathcal{D}$.

4.2.2 Alternating Two-Way Tree Automata

Alternating automata on infinite trees generalize nondeterministic tree automata and were first introduced in [MS87]. Here we describe *two-way* alternating tree automata. For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**, and, as usual, $\wedge$ has precedence over $\vee$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ (denoted $Y \models \theta$) iff assigning **true** to elements in Y and assigning **false** to elements in $X \setminus Y$ makes θ true. For a set Υ of directions, the *extension* of Υ is the set $ext(\Upsilon) = \Upsilon \cup \{\varepsilon, \uparrow\}$ (we assume that $\Upsilon \cap \{\varepsilon, \uparrow\} = \emptyset$). A *two-way alternating automaton* over Σ-labeled Υ-trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(ext(\Upsilon) \times Q)$ is the transition function, $q_0 \in Q$ is an initial state, and F is the acceptance condition.

A run of an alternating automaton $\mathcal{A}$ over a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a labeled tree $\langle T_r, r \rangle$ in which every node is labeled by an element of $\Upsilon^* \times Q$. A node in T_r, labeled by (x, q), describes a copy of the automaton that is in the state q and reads the node x of Υ^*. Note that many nodes of T_r can correspond to the same node of Υ^*; there is no one-to-one correspondence between the nodes of the run and the nodes of the input tree. The labels of a node and its successors have to satisfy the transition function. Formally, a run $\langle T_r, r \rangle$ is a Σ_r-labeled Γ-tree, for some set Γ of directions, where $\Sigma_r = \Upsilon^* \times Q$ and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\varepsilon, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, \tau(x)) = \theta$. Then there is a (possibly empty) set $S \subseteq ext(\Upsilon) \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and the following hold:
   - If $c \in \Upsilon$, then $r(\gamma \cdot y) = (c \cdot x, q')$.
   - If $c = \varepsilon$, then $r(\gamma \cdot y) = (x, q')$.
   - If $c = \uparrow$, then $x = \upsilon \cdot z$, for some $\upsilon \in \Upsilon$ and $z \in \Upsilon^*$, and $r(\gamma \cdot y) = (z, q')$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $c = \uparrow$, we require that $x \neq \varepsilon$. We extend the concatenation operator to handle $\uparrow$ (when possible). For a node $x = \upsilon \cdot z$, we denote by $\uparrow \cdot x$ the node z. For ε, the expression $\uparrow \cdot \varepsilon$ is not defined.

A run $\langle T_r, r\rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. We consider here *Büchi* and *parity* acceptance conditions [Büc62, EJ91]. A parity condition over a state set Q is a finite sequence $F = \{F_1, F_2, \ldots, F_k\}$ of subsets of Q, where $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_k = Q$. The number k of sets is called the *index* of $\mathcal{A}$. Given a run $\langle T_r, r\rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in \Upsilon^* \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies the parity condition F if there is an even $1 \leq i \leq k$ such that $inf(\pi) \cap F_i \neq \emptyset$ and $inf(\pi) \cap F_{i-1} = \emptyset$. A Büchi acceptance condition consists of a set $F_2 \subseteq Q$ and it can be viewed as a special case of a parity condition of index 3, where $F = \{\emptyset, F_2, Q\}$. Thus, a run is accepting according to the Büchi condition F_2 if every path in the run visits F_2 infinitely often. An automaton accepts a labeled tree if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. The automaton $\mathcal{A}$ is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

An automaton is 1-way if it does not use ϵ-transitions nor $\uparrow$-transitions. Formally, an automaton is 1-way if for every state $q \in Q$ and letter $\sigma \in \Sigma$ the transition $\delta(q, \sigma)$ is restricted to formulas in $B^+(\Upsilon \times Q)$. An automaton is nondeterministic if in every transition exactly one copy of the automaton is sent in every direction in Υ. Formally, an automaton is nondeterministic if for every state $q \in Q$ and letter $\sigma \in \Sigma$ there exists some set I such that $\delta(q, \sigma) = \bigvee_{i \in I} \bigwedge_{\upsilon \in \Upsilon} (s_{i,\upsilon}, \upsilon)$. Equivalently, we can describe the transition function of a nondeterministic automaton as $\delta : Q \times \Sigma \to 2^{(Q^{|\Upsilon|})}$. The tuple $\langle q_1, \ldots, q_{|\Upsilon|}\rangle \in \delta(q, \sigma)$ is equivalent to the disjunct $(q_1, \upsilon_1) \wedge \ldots \wedge (q_{|\Upsilon|}, \upsilon_{|\Upsilon|})$. In particular, a nondeterministic automaton is 1-way.

We use acronyms in $\{2, 1\} \times \{A, N\} \times \{P, B\} \times \{T, W\}$ to denote the different types of automata. The first symbol stands for the type of movement of the automaton: 2 stands for 2-way automata and 1 stands for 1-way automata (we often omit the 1). The second symbol stands for the branching mode of the automaton: A for alternating and N for nondeterministic. The third symbol stands for the type of acceptance used by the automaton: P for parity and B for Büchi, and the last symbol stands for the object the automaton is reading: T for trees and W for words. For example, a 2APT is a 2-way alternating parity tree automaton and an NPT is a 1-way nondeterministic parity tree automaton.

Theorem 4.2.1 *Given a 2APT $\mathcal{A}$ with n states and index k, we can construct an equivalent NPT whose number of states is $(nk)^{O(nk)}$ and whose index is linear in nk* [Var98]*, and we can check the nonemptiness of $\mathcal{A}$ in time $(nk)^{O((nk)^2)}$* [EJS93].

The *membership problem* of an automaton $\mathcal{A}$ and a regular tree $\langle \Upsilon^*, \tau\rangle$ is to decide whether $\langle \Upsilon^*, \tau\rangle \in \mathcal{L}(\mathcal{A})$. As described in Theorem 4.2.2 below, the membership problem can be reduced to the emptiness problem.

Theorem 4.2.2 *The membership problem of a regular tree $\langle \Upsilon^*, \tau\rangle$ and a 2APT $\mathcal{A}$ with n states and index k is solvable in time $(|\tau|nk)^{O((nk)^2)}$.*

Proof: Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F\rangle$ be a 2APT and $\langle \Upsilon^*, \tau\rangle$ be a regular tree. According to Theorem 4.2.1, we construct a 1NPT $\mathcal{N} = \langle \Sigma, P, \rho, p_0, \alpha\rangle$ that accepts the language of $\mathcal{A}$. The number of states of $\mathcal{N}$ is exponential in nk and its index is linear in nk. Let $\mathcal{D} = \langle \Upsilon, \Sigma, S, \eta, s_0, L\rangle$ be the transducer generating τ, with $\Upsilon = \{\upsilon_1, \ldots, \upsilon_d\}$.

Consider the NPT $\mathcal{N}' = \langle \{a\}, P \times S, \rho', (p_0, s_0), F\rangle$ where

$$\rho'((p, s), a) = \{\langle (p_1, s_1), \ldots, (p_d, s_d)\rangle \mid \langle p_1, \ldots, p_d\rangle \in \rho(p, L(s)) \text{ and } s_c = \eta(s, \upsilon_c)\}$$

It is easy to see that $\mathcal{L}(\mathcal{N}') \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{N})$. As $\mathcal{L}(\mathcal{N}) = \mathcal{L}(\mathcal{A})$, we are done. Note that the number of states of $\mathcal{N}'$ is $|\tau|(nk)^{O(nk)}$ and its index is linear in nk. Thus, emptiness of $\mathcal{N}'$ can be determined in time $(|\tau|nk)^{O((nk)^2)}$. □

Once we translate $\mathcal{A}$ to $\mathcal{N}$, the reduction above is similar the one described in [KVW00]. The translation of $\mathcal{A}$ to $\mathcal{N}$, however, involves an exponential blow up. In the full version of [KPV02a] we show that the membership problem for 2ABT is EXPTIME-hard. Thus, the membership problem for 2APT is EXPTIME-complete.

4.2.3 Pushdown Tree Automata

Pushdown tree automata are finite-state automata augmented by a pushdown store. Like a nondeterministic finite-state tree automaton, a nondeterministic pushdown tree automaton starts reading a tree from the root. At each node of the tree, the pushdown automaton consults the transition relation and sends independent copies of itself to each of the node's successors. Each copy has an independent pushdown store that diverges from the pushdown store of the parent. We then check what happens along every branch of the run tree and determine acceptance.

Let $\Upsilon = \{\upsilon_1, \ldots, \upsilon_d\}$. A *nondeterministic parity pushdown tree automaton* (with ϵ-transitions) over infinite Υ-trees (or *PD-NPT* for short) is $\mathcal{P} = \langle \Sigma, \Gamma, P, \rho, p_0, \alpha_0, F \rangle$, where Σ is a finite input alphabet, Γ is a finite set of pushdown symbols, P is a finite set of states, ρ is a transition function (see below), $p_0 \in P$ is an initial state, $\alpha_0 \in \Gamma^* \cdot \bot$ is an initial pushdown store content, and F is a parity condition over P.

The transition function $\rho : P \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\bot\}) \to 2^{P \times \Gamma^*} \cup 2^{(P \times \Gamma^*)^d}$ is defined such that for every state $p \in P$ and symbol $A \in \Gamma$, we have $\delta(p, a, A) \in 2^{(P \times \Gamma^*)^d}$, for $a \in \Sigma$, and $\delta(p, \epsilon, A) \in 2^{P \times \Gamma^*}$. Intuitively, when the automaton is in state p, reading a node x labeled by $a \in \Sigma$, and the pushdown store contains a word in $A \cdot \Gamma^*$, it can apply one of the following two types of transitions.

- An *ϵ-transition* in $\delta(p, \epsilon, A)$, where the automaton stays in node x. Accordingly, each ϵ-transition is a pair $(p', y) \in P \times \Gamma^*$. Once the automaton chooses a pair (p', y), it moves to state p', and updates the pushdown store by removing A and pushing y.
- An *advancing transition* in $\delta(p, a, A)$, where the automaton splits into d copies, each reading a different successor of node x. Accordingly, each advancing transition is a d-tuple $\langle (p_1, y_1), \ldots, (p_d, y_d) \rangle \in (P \times \Gamma^*)^d$. Once the automaton chooses a tuple, it splits into d copies, the ith copy moves to the node $i \cdot x$ in the input tree, changes to state p_i, and updates the pushdown store by removing A and pushing y_i.

We assume that the bottom symbol on the pushdown store is $\bot$. This symbol cannot be removed (so, when we say that the pushdown store is empty, we mean that it contains only $\bot$). Every transition that removes $\bot$ also pushes it back. Formally, if $(p', y) \in \delta(p, \epsilon, \bot)$, then $y \in \Gamma^* \cdot \bot$. Similarly, if $\langle (p_1, y_1), \ldots, (p_d, y_d) \rangle \in \delta(p, a, \bot)$, then $y_i \in \Gamma^* \cdot \bot$ for all $1 \leq i \leq d$. The symbol $\bot$ is not used in another way.

The size $|\rho|$ of the transition function is the sum of all the lengths of the words used in the function. Formally, $|\rho| = \big(\Sigma_{\langle (p_1, y_1), \ldots, (p_d, y_d) \rangle \in \rho(p,a,A)} |y_1| + \ldots + |y_d|\big) + \big(\Sigma_{(p', y) \in \rho(p, \epsilon, A)} |y|\big)$.

We note that the automata defined above assume input trees with a fixed and known branching degree, and can distinguish between the different successors of the node (say, impose a requirement only on the leftmost successor). In many cases, it is useful to consider *symmetric* tree automata [JW95], which refer to the successors of a node in a universal or an existential manner, and thus can handle trees with unknown and varying branching degrees. While symmetry is naturally defined for alternating automata, it can also be defined for nondeterministic automata [KV01], and for PD-NPT.

Example 4.2.3 *In Section 4.1, we mentioned the non-regular property "p is uniformly inevitable", namely there is some time i such that p holds at time i in all the computations. We now describe a PD-NPT for this property. We define* $\mathcal{P} = \langle 2^{\{p\}}, \{A\}, \{q_0, q_1, q_2\}, \delta, q_0, \bot, F\rangle$, *where* $F = \{\{q_0, q_1\}, \{q_0, q_1, q_2\}\}$ *is such that* q_0 *and* q_1 *have to be visited only finitely often, and the transition function is as follows.*

- $\rho(q_0, \epsilon, \bot) = \{(q_0, A\bot), (q_1, \bot)\}$,
- $\rho(q_0, \epsilon, A) = \{(q_0, AA), (q_1, A)\}$,
- $\delta(q_1, \{p\}, A) = \delta(q_1, \emptyset, A) = \langle (q_1, \epsilon), \ldots, (q_1, \epsilon)\rangle$,
- $\delta(q_1, \{p\}, \bot) = \langle (q_2, \bot), \ldots, (q_2, \bot)\rangle$, *and*
- $\delta(q_2, \epsilon, \bot) = \{(q_2, \bot)\}$.

Intuitively, $\mathcal{P}$ *starts reading the tree in state* q_0 *with empty pushdown store. It stays in state* q_0 *taking* ϵ*-transitions while pushing A's into the pushdown store. In some stage,* $\mathcal{P}$ *takes a nondeterministic choice to move to state* q_1*, from which it proceeds with advancing transitions while removing A's from the pushdown store. When the pushdown store becomes empty,* $\mathcal{P}$ *takes an advancing transition to state* q_2 *while checking that the label it reads is indeed* $\{p\}$.

A *run* of the PD-NPT $\mathcal{P}$ on an infinite tree $\langle \Upsilon^*, \tau\rangle$ is an $(\Upsilon^* \times P \times \Gamma^*)$-labeled $\mathbb{N}$-tree $\langle T_r, r\rangle$. A node $x \in T_r$ labeled by (y, p, α) represents a copy of $\mathcal{P}$ in state p, with pushdown store content α, reading node y in $\langle \Upsilon^*, \tau\rangle$. Formally, $r(\epsilon) = (\epsilon, p_0, \alpha_0)$, and for all $x \in T_r$ such that $r(x) = (y, p, A\cdot\alpha)$ one of the following holds.

- There is a unique successor $c \cdot x$ of x in T_r such that $r(c \cdot x) = (y, p', \beta \cdot \alpha)$ for some $(p', \beta) \in \delta(p, \epsilon, A)$.
- There are d successors $1 \cdot x, \ldots, d \cdot x$ of x in T_r such that for all $1 \leq c \leq d$, we have $r(c \cdot x) = (\upsilon_c \cdot y, p_c, \beta_c \cdot \alpha)$ for some $\langle (p_1, \beta_1), \ldots, (p_d, \beta_d)\rangle \in \delta(p, \tau(y), A)$.

Given a path $\pi \subseteq T_r$, we define $inf(\pi) \subseteq P$ to be such that $p \in inf(\pi)$ if and only if there are infinitely many nodes $y \in \pi$ for which $r(y) \in \Upsilon^* \times \{p\} \times \Gamma^*$. As with 2APTs, a path satisfies the parity condition $F = \{F_1, \ldots, F_k\}$ if there is an even $1 \leq i \leq k$ such that $inf(\pi) \cap F_i \neq \emptyset$ and $inf(\pi) \cap F_{i-1} = \emptyset$. A run is *accepting* if every path $\pi \subseteq T_r$ is accepting. A PD-NPT $\mathcal{P}$ accepts a tree $\langle T, \tau\rangle$ if there exists an accepting run of $\mathcal{P}$ over $\langle T, \tau\rangle$. The language of $\mathcal{P}$, denoted $\mathcal{L}(\mathcal{P})$ contains all trees accepted by $\mathcal{P}$. The PD-NPT $\mathcal{P}$ is *empty* if $\mathcal{L}(\mathcal{P}) = \emptyset$.

Harel and Raz consider only the Büchi acceptance condition (PD-NBT for short) . They showed that the emptiness problem of PD-NBT can be reduced to the emptiness problem of a

PD-NBT with one-letter input alphabet [HR94]. The parity acceptance condition is more general than the Büchi acceptance condition. The following theorem generalizes the result of [HR94] to PD-NPT.

Theorem 4.2.4 *The emptiness problem for a PD-NPT $\mathcal{P}$ with n states, index k, and input alphabet Σ, is reducible to the emptiness problem for a PD-NPT $\mathcal{P}'$ with $n \cdot |\Sigma|$ states and index k that has a one-letter input alphabet.*

Note that since our automata have ϵ-transitions, we cannot use the classical reduction to one-letter input alphabet [Rab69]. For a nondeterministic tree automaton $\mathcal{N} = \langle \Sigma, P, \rho, p_0, F \rangle$, Rabin constructs the automaton $\mathcal{N}' = \langle \{a\}, P, \rho', p_0, F \rangle$ such that for every state $p \in P$ we have $\rho'(p, a) = \bigcup_{\sigma \in \Sigma} \rho(p, \sigma)$. Thus, $\mathcal{P}'$ guesses which of the input letters $\sigma \in \Sigma$ labels the node and chooses a move in $\rho(p, \sigma)$. For automata with ϵ-transitions, we have to make sure that successor states that read the same node guess the same label for the node, and we augment the automaton $\mathcal{P}'$ with a mechanism that remembers the guessed input letter.

4.3 The Emptiness Problem for PD-NPT

In this section we give an algorithm to decide the emptiness of a PD-NPT. According to Theorem 4.2.4, we can restrict attention to PD-NPT with one-letter input alphabet. We reduce the emptiness of a PD-NPT with one-letter input alphabet $\mathcal{P}$ to the membership of a regular tree in the language of a 2APT $\mathcal{A}$. The idea behind the construction is that since the one-letter tree is homogeneous, the location of a copy of $\mathcal{P}$ in the input tree is not important. Accordingly, when a copy of $\mathcal{A}$ simulates a copy of $\mathcal{P}$, it does not care about the location on the input tree, and it has to remember only the state of the copy and the content of the pushdown store.

It is easy for a copy of $\mathcal{A}$ to remember a state of $\mathcal{P}$. How can $\mathcal{A}$ remember the content of the pushdown store? Let Γ denote the pushdown alphabet of $\mathcal{P}$. Note that the content of the pushdown store of $\mathcal{P}$ corresponds to a node in the full infinite Γ-tree. So, if $\mathcal{A}$ reads the tree Γ^*, it can refer to the location of its reading head in Γ^* as the content of the pushdown store. We would like $\mathcal{A}$ to "know" the location of its reading head in Γ^*. A straightforward way to do so is to label a node $x \in \Gamma^*$ by x. This, however, involves an infinite alphabet, and results in trees that are not regular. Since $\mathcal{P}$ does not read the entire pushdown store's content and (in each transition) it only reads the top symbol on the pushdown store, it is enough to label x by its direction.

Let $\langle \Gamma^*, \tau_\Gamma \rangle$ be the Γ labeled Γ-tree such that for every $x \in \Gamma^*$, we have $\tau_\Gamma(x) = dir(x)$. Note that $\langle \Gamma^*, \tau_\Gamma \rangle$ is a regular tree of size $|\Gamma| + 1$. We reduce the emptiness of a one-letter PD-NPT to the membership problem of $\langle \Gamma^*, \tau_\Gamma \rangle$ in the language of a 2APT. Given a PD-NPT $\mathcal{P}$ we construct a 2APT $\mathcal{A}$ such that $\mathcal{L}(\mathcal{P}) \neq \emptyset$ iff $\langle \Gamma^*, \tau_\Gamma \rangle \in \mathcal{L}(\mathcal{A})$. The 2APT memorizes a control state of the PD-NPT as part of its finite control. When it has to apply some transition of $\mathcal{P}$, it consults its finite control and the label of the tree $\langle \Gamma^*, \tau_\Gamma \rangle$. Knowing the state of $\mathcal{P}$ and the top symbol of the pushdown store, the 2APT can decide which transition of $\mathcal{P}$ to apply. Moving to a new state of $\mathcal{P}$ is done by changing the state of the 2APT. Adjusting the pushdown store's content is done by navigating to a new location in $\langle \Gamma^*, \tau_\Gamma \rangle$.

Theorem 4.3.1 *Given a one-letter PD-NPT $\mathcal{P} = \langle \{a\}, \Gamma, P, \delta, p_0, \alpha_0, F \rangle$ with n states and index k, there exists a 2APT $\mathcal{A}$ with $n \cdot |\rho|$ states and index k such that $\mathcal{L}(\mathcal{P}) \neq \emptyset$ iff $\langle \Gamma^*, \tau_\Gamma \rangle \in \mathcal{L}(\mathcal{A})$.*

Let $\Upsilon = \{v_1, \ldots, v_d\}$. Formally, given the PD-NPT $\mathcal{P} = \langle \{a\}, \Gamma, P, \delta, p_0, \alpha_0, F \rangle$, we construct the 2APT $\mathcal{A} = \langle \Gamma, Q, \eta, q_0, F' \rangle$, where

- $Q = P \times \mathit{heads}(\delta)$ where $\mathit{heads}(\delta) \subseteq \Gamma^*$ is the set of all prefixes of words $x \in \Gamma^*$ for which one of the following holds.
  - There are states $p, p_1, \ldots, p_d \in P$, words $\beta_1, \ldots, \beta_d \in \Gamma^*$, and a letter $\gamma \in \Gamma$ such that $\langle (p_1, \beta_1), \ldots, (p_d, \beta_d) \rangle \in \delta(p, \gamma, a)$ and $x = \beta_i$ for some $1 \leq i \leq d$.
  - There are states $p, p' \in P$ and a letter $\gamma \in \Gamma$ such that $(p', x) \in \delta(p, \epsilon, \gamma)$.
  - $x = \alpha_0$.

  Intuitively, when $\mathcal{A}$ visits a node $x \in \Gamma^*$ in state (p, y), it checks that $\mathcal{P}$ with initial configuration $(p, y \cdot x)$ accepts the one-letter Υ-tree. In particular, when $y = \varepsilon$, then $\mathcal{P}$ with initial configuration (p, x) needs to accept the one-letter Υ-tree.

  States of the form (p, ε) are called *action states*. From these states $\mathcal{A}$ consults δ in order to impose new requirements on $\langle \Gamma^*, \tau_\Gamma \rangle$. States of the form (p, y), for $y \in \Gamma^+$, are called *navigation states*. From these states $\mathcal{A}$ only navigates downward y to reach new action states.

- The transition function η is defined for every state $(p, x) \in P \times \mathit{heads}(\delta)$ and $A \in \Gamma$ as follows.
  - $\eta((p, \epsilon), A) = \left[\bigvee_{\substack{(t,\alpha) \in \\ \delta(p,\epsilon,A)}} (\uparrow, (t, \alpha)) \right] \vee \left[\bigvee_{\substack{\langle (t_1,\beta_1),\ldots,(t_d,\beta_d) \rangle \in \\ \delta(s,a,A)}} \bigwedge_{i=1}^{d} (\uparrow, (t_i, \beta_i)) \right]$
  - $\eta((p, \alpha \cdot B), A) = (B, (p, \alpha))$

  Thus, in action states, $\mathcal{A}$ reads the direction of the current node and applies a transition from δ. In navigation states, $\mathcal{A}$ needs to go downward to $\alpha \cdot B$, so it continues in direction B.

- $q_0 = (p_0, \alpha_0)$. Thus, in its initial state $\mathcal{A}$ checks that $\mathcal{P}$ with initial configuration (p_0, α_0) accepts the one-letter Υ-tree.

- $F' = F \times \{\epsilon\}$. Note that only action states can be accepting states.

We show that $\mathcal{A}$ accepts $\langle \Gamma^*, \tau_\Gamma \rangle$ iff $\mathcal{P}$ accepts the one letter Υ-tree. Let $\langle \Upsilon^*, \tau_a \rangle$ denote the labeled tree such that for all $x \in \Upsilon^*$, we have $\tau_a(x) = a$.

Claim 4.3.2 *$\mathcal{L}(\mathcal{P}) \neq \emptyset$ iff $\langle \Gamma^*, \tau_\Gamma \rangle \in \mathcal{L}(\mathcal{A})$.*

The proof consists of 'translating' a run tree of $\mathcal{P}$ on $\langle \Upsilon^*, \tau_a \rangle$ to a run tree of $\mathcal{A}$ on $\langle \Gamma^*, \tau_\Gamma \rangle$ and vice versa. Every transition that $\mathcal{P}$ takes in this run is translated to the appropriate series of transitions of $\mathcal{A}$. The skeleton of the two run trees is identical, the alternating automaton takes a series of moves (the navigation states leading to an action state) to mimic a single move of the pushdown automaton. An action state in the run of $\mathcal{A}$ in location w in the tree Γ^* corresponds to a state of $\mathcal{P}$ with w on the pushdown store.

Proof: We have to work with four different trees. We have a PD-NPT $\mathcal{P}$ and a 2APT $\mathcal{A}$, each has an input tree and a run tree. We introduce a special notation as follows.

1. Let $T_i^{PD} = \langle \Upsilon^*, \tau_a \rangle$ denote the input tree read by $\mathcal{P}$.
2. Let $T_i^{A} = \langle \Gamma^*, \tau_\Gamma \rangle$ denote the input tree read by $\mathcal{A}$.
3. Let $\langle T_r^{PD}, r^{PD} \rangle$ denote the run tree of $\mathcal{P}$ and its labeling.
4. Let $\langle T_r^{A}, r^{A} \rangle$ denote the run tree of $\mathcal{A}$ and its labeling.

Assume first that $T_i^{PD} \in \mathcal{L}(\mathcal{P})$. Then, there exists an accepting run $\langle T_r^{PD}, r^{PD} \rangle$ of $\mathcal{P}$ on T_i^{PD}. Given T_i^{PD} and $\langle T_r^{PD}, r^{PD} \rangle$, we have to construct an accepting run tree $\langle T_r^{A}, r^{A} \rangle$ of $\mathcal{A}$ on T_i^{A}. Consider a node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (y, (p, \gamma))$. Recall that x stands for a copy of $\mathcal{P}$ in state p with pushdown store content γ, reading node $y \in T_i^{PD}$. We associate with x a node $x' \in T_r^{A}$ labeled by $r^{A}(x') = (\gamma, (p, \epsilon))$. Recall that x' stands for a copy of $\mathcal{A}$ in state (p, ϵ), reading node $\gamma \in T_i^{A}$. Both nodes are labeled by the state p of $\mathcal{P}$. The pushdown store content of $\mathcal{P}$ is the location of $\mathcal{A}$ in T_i^{A}.

We prove by induction that we can build $\langle T_r^{A}, r^{A} \rangle$ in such a way. We start from the root $\epsilon \in T_r^{PD}$ labeled by $r^{PD}(\epsilon) = (\epsilon, (p_0, \alpha_0))$. The root $\epsilon \in T_r^{A}$ is labeled by $r^{A}(\epsilon) = (\epsilon, (p_0, \alpha_0))$. The behavior of the 2APT is deterministic until it reaches the next action state. Thus, there is some $x' \in T_r^{A}$ labeled by $r^{A}(x') = (\alpha_0, (p_0, \epsilon))$ which serves as the base case for the induction.

Given a node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (y, (p, A \cdot \alpha))$, by the induction assumption it is associated with a node $x' \in T_r^{A}$ labeled by $r^{A}(x') = (A \cdot \alpha, (p, \epsilon))$.

Suppose x has one successor $c \cdot x$ labeled by $r^{PD}(c \cdot x) = (y, (p', \beta \cdot \alpha))$, that resulted from the transition $(p', \beta) \in \delta(p, \epsilon, A)$. Then there is a disjunct $(\uparrow, (p', \beta))$ that appears in $\eta(p, A)$. We add $c \cdot x'$ to T_r^{A}, the unique successor of x', and label it $r^{A}(c \cdot x') = (\alpha, (p', \beta))$. Obviously, this satisfies η. Again the behavior below $c \cdot x'$ is deterministic until reaching a node $z \cdot x' \in T_r^{A}$ labeled by $r^{A}(z \cdot x') = (\beta \cdot \alpha, (p, \epsilon))$.

Suppose x has d successors $c_1 \cdot x, \ldots, c_d \cdot x \in T_r^{PD}$ labeled by $r^{PD}(c_i \cdot x) = (y_i, (p_i, \beta_i \cdot \alpha))$, that resulted from the transition $\langle (p_1, \beta_1), \ldots, (p_d, \beta_d) \rangle \in \delta(p, a, A)$. Then there is a disjunct $\bigwedge_{i=1}^{d} (\uparrow, (p_i, \beta_i))$ in $\eta(p, A)$. We add $c_1 \cdot x', \ldots c_d \cdot x'$ to T_r^{A} as the successors of x', and label them $r^{A}(c_i \cdot x') = (\gamma, (p_i, \beta_i))$. Obviously, this satisfies η. The behavior of each path below $c_i \cdot x'$ is deterministic until reaching some node $z_i \cdot x' \in T_r^{A}$ labeled by $r^{A}(z_i \cdot x') = (\beta_i \cdot \alpha, (p_i, \epsilon))$.

We have to show now that (T_r^{A}, r^{A}) is accepting. Consider an infinite path $\pi' \subseteq T_r^{A}$. Clearly, π' visits infinitely many action states. Every action state and the node it labels, is associated by the construction with a node in T_r^{PD}. It is quite clear that this sequence of nodes in T_r^{PD} forms a path $\pi \subseteq T_r^{PD}$. Hence if π satisfies F, it is also the case that π' satisfies F'.

Assume now that $\mathcal{A}$ accepts $\langle \Gamma^*, \tau_\Gamma \rangle$. There exists an accepting run $\langle T_r^{A}, r^{A} \rangle$ of $\mathcal{A}$ on $\langle \Gamma^*, \tau_\Gamma \rangle$. We construct an accepting run $\langle T_r^{PD}, r^{PD} \rangle$ of $\mathcal{P}$ on T_i^{PD}. We convert the set of nodes in T_r^{A} labeled by action states of $\mathcal{A}$ to the tree T_r^{PD}. We update the location of $\mathcal{P}$ in Υ^* according to the number of successors of each action state. One successor matches an ϵ-move and d successors match a forward move.

Formally, we assume by induction that every node $x' \in T_r^{A}$ labeled by $r^{A}(x') = (\alpha, (p, \epsilon))$ is associated with some node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (y, (p, \alpha))$ for some node $y \in T_i^{PD}$. As

before the root ϵ of T_r^A is labeled by $r^A(\epsilon) = (\epsilon, (p_0, \alpha_0))$. It has some descendant $x' \in T_r^A$ labeled by $r^A(x') = (\alpha_0, (p_0, \epsilon))$. We label the root $\epsilon \in T_r^{PD}$ by $r^{PD}(\epsilon) = (\epsilon, (p_0, \alpha_0))$. This serves as the induction base.

Given some node $x' \in T_r^A$ labeled by $r^A(x') = (A \cdot \alpha, (p, \epsilon))$, by induction assumption there is a node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (y, (p, A \cdot \alpha))$.

Suppose x' has one successor $c \cdot x'$ labeled by $r^A(c \cdot x') = (\alpha, (p', \beta))$, that resulted from the disjunct $(\uparrow, (p', \beta))$ that appears in $\eta(p, A)$. Again, there is some descendant $z \cdot x'$ of $c \cdot x'$ that is an action state labeled by $r(z \cdot x') = (\beta \cdot \alpha, (p', \epsilon))$. We know that $(p', \beta) \in \delta(p, \epsilon, A)$, we add $c \cdot x$ to T_r^{PD}, a unique successor of x, and label it $r^{PD}(c \cdot x) = (y, (p', \beta \cdot \alpha))$. Note that $c \cdot x$ and x read the same node $y \in T_i^{PD}$.

Suppose x' has d successors $c_1 \cdot x', \ldots, c_d \cdot x' \in T_r^A$ labeled by $r^A(c_i \cdot x') = (\alpha, (p_i, \beta_i))$, that resulted from the disjunct $\bigwedge_{i=1}^{d}(\uparrow, (p_i, \beta_i))$ in $\eta(p, A)$. Each one of the nodes $c_i \cdot x'$ has a descendant $z_i \cdot x'$, that is labeled by the action state $r(z_i \cdot x') = (\beta \cdot \alpha, (p_i, \epsilon))$. We know that $\langle (p_1, \beta_1), \ldots, (p_d, \beta_d) \rangle \in \delta(p, a, A)$. We add $c_1 \cdot x, \ldots c_d \cdot x$ to T_r^{PD} as the successors of x, and label them $r^{PD}(c_i \cdot x) = (v_i \cdot y, (p_i, \beta_i \cdot \alpha))$. Obviously, this satisfies δ.

Again every path in T_r^{PD} is associated with the action states along a path in T_r^A. We conclude that $\langle T_r^{PD}, r^{PD} \rangle$ satisfies F. □

Combining Theorem 4.3.1 with Theorems 4.2.2 and 4.2.4, we get the following.

Corollary 4.3.3 *The emptiness problem for a PD-NPT with n states, index k, and transition function ρ can be solved in time exponential in $nk \cdot |\rho|$.*

In Appendix 4.A we show that a reduction in the other direction is also possible. We show that the membership problem for 2APT can be solved by the emptiness problem of PD-APT (with one letter input alphabet).

Remark 4.3.4 Harel and Raz [HR94] show that the emptiness of stack automata on infinite trees is also decidable. Stack automata can read the entire contents of the stack but can change the content only when standing on the top of the stack. They give a doubly exponential reduction from the emptiness problem of stack automata to the emptiness problem of pushdown automata. As their emptiness of pushdown automata is triple exponential, it induces a five fold exponential algorithm for the emptiness of stack automata on infinite trees that use the Büchi acceptance condition. Thus, our emptiness algorithm induces a triple exponential algorithm for the emptiness of Büchi stack automata. In [PV03] we extend the reduction of [HR94] to parity stack automata and show that using our techniques, the emptiness of parity stack automata can be solved in doubly exponential time. □

4.4 Model-Checking Pushdown Specifications of Finite-State Systems

The *model-checking* problem is to decide whether a given system $\mathcal{S}$ satisfies a specification $\mathcal{P}$. In this section we consider the case where the system is finite state and the specification is a PD-NPT. In order to solve the model-checking problem we combine the system with the PD-NPT and get a PD-NPT whose language is empty iff the system satisfies the specification.

We use *labeled transition graphs* to represent finite-state systems. A labeled transition graph is a quadruple $\mathcal{S} = \langle W, Act, R, w_0 \rangle$, where W is a (possibly infinite) set of states, Act is a finite set of actions, $R \subseteq W \times Act \times W$ is a labeled transition relation, and $w_0 \in W$ is an initial state. We assume that the transition relation R is total (i.e. for every state there exists some action a and some state w' such that $R(w, a, w')$). When $R(w, a, w')$, we say that w' is an *a-successor* of w, and w is an *a-predecessor* of w'. For a state $w \in W$, we denote by $\mathcal{S}^w = \langle W, Act, R, w \rangle$, the system $\mathcal{S}$ with w as its initial state. A finite-state system is given as a labeled transition graph with a finite set of states. The *unwinding* of $\mathcal{S}$ from state $w \in W$ induces an infinite tree $T_{\mathcal{S}}$. Every node of the tree $T_{\mathcal{S}}$ is associated with some state $w' \in W$, the root of $T_{\mathcal{S}}$ is associated with state w. A node $x \in T_{\mathcal{S}}$ associated with $w' \in W$ has $|\{w'' \mid \exists a \in Act \text{ s.t. } R(w', a, w'')\}|$ successors, each associated with a successor w'' of w' and labeled by the action a such that $R(w', a, w'')$. The root of $T_{\mathcal{S}}$ is labeled by $\perp \notin Act$. As R is total, $T_{\mathcal{S}}$ is infinite. We say that a system $\mathcal{S}$ *satisfies* a tree automaton $\mathcal{P}$ over the alphabet Act if $T_{\mathcal{S}}$ is accepted by $\mathcal{P}$.[2]

The unwinding of a finite labeled transition graph $\mathcal{S}$ results in a regular tree. In order to determine whether $\mathcal{S}$ satisfies $\mathcal{P}$, we have to solve the membership problem of regular trees in the language of a PD-NPT. We reduce the membership problem to the emptiness problem by a construction similar to the one in the proof of Theorem 4.2.2. We construct a PD-NPT that either accepts $T_{\mathcal{S}}$ or is empty, and then check its emptiness.

Theorem 4.4.1 *Given a finite labeled transition graph $\mathcal{S}$ with m states and a PD-NPT $\mathcal{P}$ with n states, index k, and transition function ρ, the model-checking problem of $\mathcal{S}$ with respect to $\mathcal{P}$ is solvable in time exponential in $mnk \cdot |\rho|$.*

Given a PD-NPT $\mathcal{P} = \langle Act, \Gamma, P, \rho, p_0, \alpha_0, F \rangle$ and a labeled transition graph $\mathcal{S} = \langle W, Act, R, w_0 \rangle$ where $|W|$ is finite, we construct the PD-NPT $\mathcal{P}' = \langle \{b\}, \Gamma, P \times Act \times W, \rho', (p_0, \perp, w_0), \alpha_0, F' \rangle$ that is the product of the two. The states of $\mathcal{P}'$ consist of triplets of states of $\mathcal{P}$, actions of $\mathcal{S}$, and states of $\mathcal{S}$. The acceptance condition F' is $F \times Act \times W$, where we replace each set $F_i \in F$ by the set $F_i \times Act \times W$. The transition function ρ' maps a triplet (p, a, w) to all the ϵ-successors of p tagged again by a and w and to all the "a-successors" of p tagged by successors of w and the actions taken to get to them. For technical convenience, we assume that the branching degree of $\mathcal{S}$ is uniform and equivalent to the branching degree of the trees read by $\mathcal{P}$. Modifying the algorithm to systems with nonuniform branching degree is not too complicated. Formally, we have the following.

- $\rho'((p, a, w), \epsilon, A) = \{\langle (p', a, w), \alpha \rangle \mid \langle p, \alpha \rangle \in \rho(p, \epsilon, A)\}$.
- $\rho'((p, a, w), b, A) = \{\langle \langle (p_1, a_1, w_1), \alpha_1 \rangle, \ldots, \langle (p_d, a_d, w_d), \alpha_d \rangle \rangle \mid$
  $\langle \langle p_1, \alpha_1 \rangle, \ldots, \langle p_d, \alpha_d \rangle \rangle \in \rho(p, a, A)$ and
  $\{(w, a_1, w_1), \ldots, (w, a_d, w_d)\}$ is the set of transitions from $w\}$.

It is not hard to see that $\mathcal{P}'$ accepts some tree iff $\mathcal{P}$ accepts $T_{\mathcal{S}}$, the unwinding of $\mathcal{S}$.

Remark 4.4.2 By having PD-NPT as our specification formalism, we follow the branching-time paradigm to specification and verification, where the specification describes allowed computation

[2] There is a slight technical difficulty as in our formalism PD-NPT run on trees with a uniform branching degrees, while labeled transition graphs are not required to have a uniform outdegree. This difficulty can be finessed by allowing automata on non-uniform trees, as, for example, in [KVW00].

trees and a system is correct if its (single) computation tree is allowed. Alternatively, in the linear-time paradigm, the specification describes the allowed linear computations, and the system is correct if all its computations are allowed. When the system is nondeterministic, it may have many computations, and we have to check them all. Thus, while model checking in the branching-time paradigm corresponds to membership checking, model checking in the linear-time paradigm corresponds to checking language containment.

Pushdown specification formalisms are helpful also in the linear-time paradigm [SCFG84]. For example, one can use pushdown word automata to specify unbounded LIFO buffers. Nevertheless, since the containment problem of regular languages in context-free languages is undecidable [HMU00], using pushdown word automata as a specification formalism leads to an undecidable model-checking problem even for finite-state systems.

The branching-time paradigm is more general than the linear-time paradigm, in the sense that we can view a (universally quantified) linear-time specification as a branching-time specification [Lam80, Pnu85]. This does not contradict the fact that model checking of pushdown specification is decidable in the branching-time paradigm. Indeed, a translation of a nondeterministic pushdown word automaton that recognizes a language L into a nondeterministic pushdown tree automata that recognizes the language of all the trees derived by L (that is, trees all of whose paths are in L) is not always possible. For cases where such a translation is possible (in particular, when the pushdown word automaton is deterministic or belongs to a class that can be determinized), linear model checking is decidable. This is reminiscent of the situation with dense-time temporal logic, where model checking with respect to linear-time specifications is undecidable, while model checking with respect to branching-time specifications is decidable, cf. [ACD93]. □

Remark 4.4.3 Unlike the case of regular tree automata, it can be proved that *alternating* pushdown automata are strictly more expressive than nondeterministic pushdown automaton. For example, it is easy to define a pushdown alternating automaton over words that recognizes the non-context-free language $\{a^i b^i c^i : i \geq 1\}$. Indeed, the automaton can send two copies, one for comparing the number of a's with b's, and one for comparing the number of b's with c's. A similar argument shows that alternating pushdown tree automata are stronger than nondeterministic pushdown tree automata. It is well known that the universality problem for nondeterministic pushdown word automata is undecidable [HMU00]. It is simple to reduce the universality problem of a nondeterministic automaton to the emptiness problem of an alternating automaton. Thus, the emptiness of alternating pushdown automata is undecidable.

On the other hand, as studied in [KVW00], the membership problem for alternating automata is not harder than the one for nondeterministic automata. This observation does not change when pushdown automata are involved. In particular, it is easy to extend Theorem 4.3.1 to alternating automata (without changing the blow up), and to extend the model-checking algorithm described above to the stronger framework of alternating pushdown automata. □

4.5 Model-Checking Pushdown Specifications of Context-Free Systems

In this section we show that the decidability results of Section 4.4 cannot be extended to context-free systems. We show that the model checking problem for context-free systems is undecidable already for pushdown path automata, which are a special case of pushdown tree automata.

We first define context-free systems and pushdown path automata. Again we use labeled transition graphs. This time with an infinite number of states.

A *rewrite system* is a quadruple $\mathcal{R} = \langle V, Act, R, x_0 \rangle$, where V is a finite alphabet, Act is a finite set of actions, R maps each action a to a finite set of rewrite rules, to be defined below, and $x_0 \in V^*$ is an initial word. Intuitively, $R(a)$ describes the possible rules that can be applied by taking the action a. We consider here *context-free* rewrite systems. Each rewrite rule is a pair $\langle A, x \rangle \in V \times V^*$. We refer to rewrite rules in $R(a)$ as *a-rules*. As before, every set $R(a)$ is finite.

The rewrite system $\mathcal{R}$ induces the labeled transition graph $G_{\mathcal{R}} = \langle V^*, Act, \rho_{\mathcal{R}}, x_0 \rangle$, where $\langle x, a, y \rangle \in \rho_{\mathcal{R}}$ if there is a rewrite rule in $R(a)$ whose application on x results in y. In particular, when $\mathcal{R}$ is a context-free rewrite system, then $\rho_{\mathcal{R}}(A \cdot y, a, x \cdot y)$ if $\langle A, x \rangle \in R(a)$. A labeled transition graph that is induced by a context-free rewrite system is called a *context-free graph*.

Consider a labeled transition graph $G = \langle S, Act, \rho, s_0 \rangle$. A *nondeterministic pushdown path automaton* on labeled transition graphs is a tuple $\mathcal{P} = \langle Act, \Gamma, P, \delta, p_0, \alpha_0, F \rangle$, where Γ, P, p_0, and α_0 are as in nondeterministic pushdown automata on trees, Act is a set of actions (the automaton's alphabet), and $\delta : P \times Act \times \Gamma \to 2^{P \times \Gamma^*}$ is the transition function. We consider the simpler case where F is a Büchi acceptance condition. Intuitively, when $\mathcal{P}$ is in state p with $A \cdot \alpha$ on the pushdown store and it reads a state s of G, the automaton $\mathcal{P}$ chooses an atom $\langle p', \beta \rangle \in \delta(p, a, A)$ and moves to some a-successor of s in state p' with pushdown store $\beta \cdot \alpha$. Again we assume that the first symbol in α_0 is $\perp$, and that $\perp$ cannot be removed from the pushdown store.

Like a run of a nondeterministic pushdown automaton on words, a run of a path automaton over a labeled transition graph $G = \langle S, Act, \rho, s_0 \rangle$ is an infinite word in $(S \times P \times \Gamma^*)^\omega$. A letter (s, p, α), describes that the automaton is in state p of $\mathcal{P}$ with pushdown store content α reading state s of G. Formally, a run is an infinite sequence $(s_0, p_0, \alpha_0), (s_1, p_1, \alpha_1), \ldots \in (S \times P \times \Gamma^*)^\omega$ as follows.

- s_0 is the initial state of G, p_0 is the initial state of $\mathcal{P}$, and α_0 is the initial pushdown store content.
- For every $i \geq 0$ there exists some $a \in Act$ such that s_{i+1} is an a-successor of s_i and if $\alpha_i = A \cdot \alpha$ then $(p_{i+1}, \beta) \in \delta(p_i, a, A)$ and $\alpha_{i+1} = \beta \cdot \alpha$.

A run r is *accepting* if it satisfies the acceptance condition. The graph G is accepted by $\mathcal{P}$ if there is an accepting run on it. Let $\mathcal{L}(\mathcal{P})$ denote the set of graphs accepted by $\mathcal{P}$.

We use PD-NBP (pushdown nondeterministic Büchi path automata) as our specification language. We say that a labeled transition graph G satisfies a PD-NBP $\mathcal{P}$, denoted $G \models \mathcal{P}$, if $\mathcal{P}$ accepts G.

Theorem 4.5.1 *The model-checking problem for context-free systems and pushdown path automata is undecidable.*

Proof: We do a reduction from the halting problem for two-counter machines, shown to be undecidable in [Min67], to the problem of whether a context-free graph satisfies a specification given by a PD-NBP. In order to simulate the two-counter machine by the context-free system and the PD-NBP, we use the state of the context-free system (a word in V^*) to maintain the value of the first counter, and we use the pushdown store of the PD-NBP to maintain the value of the second counter. In order to simulate the two counters, we have to be able to check whether the value of

each counter is zero or not, and to increase and decrease the value of each counter. Handling of the second counter (maintained by the pushdown store of the path automaton) is straightforward: the path automaton can check whether its pushdown store is empty or not, can push one letter into the pushdown store, and can pop one letter from the pushdown store.

Handling of the first counter (maintained by the state of the context-free system) is a bit more complicated. The context-free system $\mathcal{S}$ has $V = \{A\}$, and its initial state is $\perp$. The rewrite rules of $\mathcal{S}$ are such that all the reachable states of $\mathcal{S}$ are in $A^* \cdot \perp$. The system $\mathcal{S}$ has five possible actions (which are also read by the PD-NBP $\mathcal{P}$): $push, pop, idle, empty\_push$, and $empty\_idle$. From the state $\perp$, the system $\mathcal{S}$ may apply the actions $empty\_push$ and $empty\_idle$, thus signaling to the specification that its counter is zero. From a state in $A^+ \cdot \perp$, the system $\mathcal{S}$ may apply the actions $push$, pop, or $idle$. The value of the first counter is simulated by the (number of A's in the) location of the PD-NBP $\mathcal{P}$ on the context-free graph. In order to apply a transition of M from a configuration in which the first counter equals zero, $\mathcal{P}$ tries to read the actions $empty\_push$ or $empty\_idle$. In order to increase the first counter, $\mathcal{P}$ reads the action $push$ (or $empty\_push$). Decreasing the counter and leaving it unchanged is similar. The path automaton $\mathcal{P}$ memorizes the state of M in its finite control. It accepts if it gets to an accepting state of M.

We first define two-counter machines. A two counter machine is $M = \langle S, \mapsto, F_{acc}, F_{rej} \rangle$, where S is a set of states, and $F_{acc} \subseteq S$ and $F_{rej} \subseteq S$ are disjoint sets of accepting and rejecting states, respectively. We assume that once M reaches an accepting or rejecting state, it loops there forever. A *configuration* of M is a triplet $\langle s, c_1, c_2 \rangle \in S \times \mathbb{N} \times \mathbb{N}$, indicating the state of the machine and the values of the two counters. The transition function $\mapsto$: $S \times \{zero, not\_zero\} \times \{zero, not\_zero\} \to 2^{S \times \{inc,dec,idle\} \times \{inc,dec,idle\}}$ maps a representation of a configuration (where the values of the counters are replaced by flags indicating whether they are equal to zero) into possible transitions of the machine, where an action involves a move to a new state and possible updates (increase or decrease) to the counters. We write $(s, v_1, v_2) \mapsto (s', d_1, d_2)$ for $(s', d_1, d_2) \in \mapsto (s, v_1, v_2)$.

The context-free system is $\mathcal{S} = \langle \{a, \perp\}, Act, T, \perp \rangle$, with Act as described above, and the following rewrite rules.

- $T(empty\_push) = \langle \perp, a\perp \rangle$. Signal that the counter is zero and add a to the state.
- $T(empty\_idle) = \langle \perp, \perp \rangle$. Signal that the counter is zero and leave the state unchanged.
- $T(push) = \langle a, aa \rangle$. Signal that the counter is not zero and add a to the state.
- $T(pop) = \langle a, \epsilon \rangle$. Signal that the counter is not zero and remove a from the state.
- $T(idle) = \langle a, a \rangle$. Signal that the counter is not zero and leave the state unchanged.

The path automaton $\mathcal{P}$ mimics M. Formally, $\mathcal{P} = \langle Act, \{a\}, S, \delta, s_0, \perp, F_{acc} \cup F_{rej} \rangle$, where the transition function δ is induced by the transition relation $\mapsto$ of M as follows. If $(s, v_1, v_2) \mapsto (s', d_1, d_2)$ then $(s', \alpha) \in \delta(s, a, B)$, where

- If $v_1 = empty$, then $a \in \{empty\_push, empty\_idle\}$. Otherwise, $a \in \{push, pop, idle\}$.
- If $v_2 = empty$, then $B = \perp$. Otherwise $B = A$.
- If $d_1 = inc$ then $a \in \{empty\_push, push\}$, if $d_1 = dec$ then $a = pop$, and if $d_1 = idle$ then $a \in \{empty\_idle, idle\}$.

- If $d_2 = inc$, then $\alpha \in \{AA, A\bot\}$, if $d_2 = dec$, then $\alpha = \epsilon$, and if $d_2 = idle$, then $\alpha \in \{A, \bot\}$.

It is not too difficult to see that $G_{\mathcal{S}} \models \mathcal{P}$ iff M terminates. □

We note that path automata are indeed weaker than tree automata. Indeed, a PD-NBT can simulate a PD-NBP by sending copies in accepting sinks to all directions but the direction to which the PD-NBP chooses to go. It follows that the model checking problem for context-free systems and PD-NPT is also undecidable.

Remark 4.5.2 In [AEM04], Alur et al. introduce the logic CARET. CARET is a linear temporal logic that can specify non-regular properties. Model-checking CARET with respect to pushdown systems is decidable in exponential time [AEM04]. We note that CARET is less expressive than pushdown automata. The study of CARET inspired Alur et al. to introduce *visibly pushdown languages*, which are a subset of the context-free languages, for which model checking with respect to pushdown systems is decidable [AM04]. □

Bibliography

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. 10th International Conference on Tools and Algorithms For the Construction and Analysis of Systems*, volume 2725 of *Lecture Notes in Computer Science*, pages 67–79, Barcelona, Spain, April 2004. Springer-Verlag.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. 36th ACM Symposium on Theory of Computing*. ACM, ACM press, 2004.

[BCMS00] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. Unpublished manuscript, 2000.

[BE96] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1996.

[BEH95] A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *Proc. 10th annual IEEE Symposium on Logic in Computer Science*, pages 123–133, San Diego, CA, USA, June 1995. IEEE computer society press.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, July 1997. Springer-Verlag.

[BER94] A. Bouajjani, R. Echahed, and R. Robbana. Verification of nonregular temporal properties for context-free processes. In *Proc. 5th International Conference on Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 81–97, Uppsala, Sweden, 1994. Springer-Verlag.

[BLM01] P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessors using satisfiability solvers. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.

[BQ96] O. Burkart and Y.-M. Quemener. Model checking of infinite graphs defined by graph grammars. In *Proc. 1st International workshop on verification of infinite states systems*, volume 6 of *ENTCS*, page 15. Elsevier, 1996.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic J. Comut.*, 2:89–125, 1995.

[BS99] O. Burkart and B. Steffen. Model checking the full modal μ-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Bur97a] O. Burkart. Automatic verification of sequential infinite-state processes. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Lecture Notes in Computer Science*, volume 1354. Springer-Verlag, 1997.

[Bur97b] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. In F. Moller, editor, *Proc. 2nd International workshop on verification of infinite states systems*, 1997.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 1996.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer-Verlag, 2001.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, San Juan, October 1991.

[EJS93] E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Comptuer Science*, pages 385–396, Elounda, Crete, June 1993. Springer-Verlag.

[Eme87] E.A. Emerson. Uniform inevitability is tree automaton ineffable. *Information Processing Letters*, 24(2):77–79, January 1987.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown automata. In F. Moller, editor, *Proc. 2nd International Workshop on Verification of Infinite States Systems*, 1997.

[HMU00] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, 2000.

[HR94] D. Harel and D. Raz. Deciding emptiness for stack automata on infinite trees. *Information and Computation*, 113(2):278–299, September 1994.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *Proc. 20th International Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 552–562. Springer-Verlag, 1995.

[KPV02a] O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 2002.

[KPV02b] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *Proc. 9th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 2514 of *Lecture Notes in Computer Science*, pages 262–277. Springer-Verlag, 2002.

[KV00] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2000.

[KV01] O. Kupferman and M.Y. Vardi. On bounded specifications. In *Proc. 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, pages 24–38. Springer-Verlag, 2001.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Lam80] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 174–185, January 1980.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[Min67] M.L. Minsky. *Computation: Finite and Infinite Machines.* Prentice Hall, London, 1 edition, 1967.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[PI95] W. Peng and S. P. Iyer. A new typee of pushdown automata on infinite tree. *International Journal of Foundations of Computer Science*, 6(2):169–186, June 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th International Colloquium on Automata, Languages and Programming*, volume 194, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.

[PV03] N. Piterman and M. Vardi. Micro-macro stack systems: A new frontier of decidability for sequential systems. In *18th IEEE Symposium on Logic in Computer Science*, pages 381–390, Ottawa, Canada, June 2003. IEEE, IEEE press.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[SCFG84] A. Sistla, E.M. Clarke, N. Francez, and Y. Gurevich. Can message buffers be axiomatized in linear temporal logic. *Information and Control*, 63(1/2):88–112, 1984.

[Tho01] W. Thomas. A short introduction to infinite automata. In *Proc. 5th. international conference on Developments in Language Theory*, volume 2295 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, July 2001.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th International Coll. on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer-Verlag, Berlin, July 1998.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer-Verlag, 1996.

[Wal01] I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, 2001.

4.A The Membership Problem for 2APT

In this section we show that we can also reduce the membership problem of 2APT to the emptiness problem of PD-NPT, thus the two problems are polynomially equivalent. The idea behind our construction is that the store of the PD-NPT can represent the location of the 2APT on the regular tree. A move of the 2APT is simulated by the PD-NPT changing the pushdown store.

In order to avoid a complex translation of the transition function of the 2APT, we use PD-APT with 1-letter input alphabet (i.e., alternating instead of nondeterministic). In [KVW00], Kupferman et al. show that emptiness of APT with 1-letter input alphabet and emptiness of NPT are intereducible in linear time and logarithmic space. It is quite simple to extend this result to pushdown automata.[3] As mentioned in Section 4.3, our reduction from emptiness of PD-NPT to membership of 2APT can be extended to handle PD-APT with 1-letter input alphabet.

Theorem 4.A.1 *Given a 2APT $\mathcal{A}$ with n states and index k and a regular tree $\langle \Upsilon^*, \tau \rangle$, there exists a PD-APT $\mathcal{P}$ over one letter input alphabet with n states and index k such that $\mathcal{L}(\mathcal{P}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{A})$.*

Proof: Let $\mathcal{A} = \langle \Sigma, Q, \eta, q_0, F \rangle$ be a 2APT over Υ-trees and let $\mathcal{D} = \langle \Sigma, \Upsilon, D, \rho, d_0, L \rangle$ be the transducer inducing the labeling of $\langle \Upsilon^*, \tau \rangle$. Assume, wlog, that d_0 does not have incoming transitions. Consider the PD-APT $\mathcal{P} = \langle \{a\}, \Upsilon \times D, Q, \delta, q_0, (\bot, d_0), F' \rangle$ where we use $(\bot, d_0)$ as the store bottom symbol and the transition function $\delta(q, a, (\upsilon, d))$ is obtained from $\eta(q, L(d))$ by replacing an atom (c, q) by

- (q, ϵ) if $c = \uparrow$. That is, a move of $\mathcal{A}$ towards the root is simulated by a pop of $\mathcal{P}$.
- $(q, (\upsilon, d))$ if $c = \varepsilon$. That is, when $\mathcal{A}$ stays on the same node, $\mathcal{P}$ does not change the store content.
- $(q, (c, \rho(d, c))(\upsilon, d))$ if $c \in \Upsilon$. That is, when $\mathcal{A}$ goes in direction c, $\mathcal{P}$ extends its pushdown store by the direction c and a state of $\mathcal{D}$.

[3] The construction in [KVW00] transforms an APT with n states into an NPT reading trees with branching degree n. The transition of the NPT is obtained from the transition of the APT by replacing an atom q_i with the atom (i, q_i). The resulting automaton in nondeterministic.

In our case, handling a 2APT with n states reading trees of branching degree m, requires a PD-NPT reading trees with branching degree $(2 + m)n$. We need a different direction for every state and every possible direction of the 2APT. Alternatively, we can construct a PD-APT with 1-letter input alphabet, and then convert it into a PD-NPT. The PD-NPT requires trees with branching degree that depends on the number of states of the PD-APT and the number of possible changes to the pushdown store.

A configuration of $\mathcal{P}$ consists of a state of $\mathcal{A}$ together with a sequence of letters from Υ and states of $\mathcal{D}$. The sequence stored on the pushdown store is in fact a location in Υ^* with the run of $\mathcal{D}$ on that location[4]. We claim that $\mathcal{L}(\mathcal{P})$ is empty iff $\langle \Upsilon^*, \tau \rangle$ is accepted by $\mathcal{A}$.

Claim 4.A.2 *$\mathcal{L}(\mathcal{P}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{A})$.*

Proof: The proof consists of 'translating' a run tree of $\mathcal{P}$ on a^ω to a run tree of $\mathcal{A}$ on $\langle \Upsilon^*, \tau \rangle$ and vice versa. Every move of one automaton corresponds to a move of the other automaton with the pushdown store corresponding to the location of the 2APT in $\langle \Upsilon^*, \tau \rangle$.

We have to work with three different trees. The PD-APT $\mathcal{P}$ reads the word a^ω and has an accepting run tree on it. The 2APT has an input tree and an accepting run tree. We introduce a special notation as follows.

1. Let $T_i^A = \langle \Upsilon^*, \tau \rangle$ denote the input tree read by $\mathcal{A}$.
2. Let $\langle T_r^A, r^A \rangle$ denote the run tree of $\mathcal{A}$ and its labeling.
3. Let $\langle T_r^{PD}, r^{PD} \rangle$ denote the run tree of $\mathcal{P}$ and its labeling.

For a sequence $\gamma \in (\Upsilon \times D)^*$ let $\gamma\Downarrow_1 \in \Upsilon^*$ denote the projection of γ on the letters in Υ (with $\perp$ removed) and $\gamma\Downarrow_2 \in D^*$ denote the reverse of the projection of γ on the states in D. Let $\alpha = \upsilon_1, \ldots, \upsilon_n$ be some word in Υ^* and $r_{\mathcal{D}}(\alpha) = d_0, \ldots, d_n$ be the unique run of $\mathcal{D}$ on the reverse of α. We denote by $\alpha \times r_{\mathcal{D}}(\alpha)$ the unique word $\gamma_0, \ldots, \gamma_n$ in $(\Upsilon \times D)^*$ such that $\gamma_n = (\perp, d_0)$ and $\gamma_i = (\upsilon_{i+1}, d_{n-i})$. In the case that $\alpha = \varepsilon$, we set $\alpha \times r_{\mathcal{D}}(\alpha) = (\perp, d_0)$.

Assume first that $a^\omega \in \mathcal{L}(\mathcal{P})$. Then, there exists an accepting run $\langle T_r^{PD}, r^{PD} \rangle$ of $\mathcal{P}$ on a^ω. Given a^ω and $\langle T_r^{PD}, r^{PD} \rangle$, we have to construct an accepting run tree $\langle T_r^A, r^A \rangle$ of $\mathcal{A}$ on T_i^A. Consider a node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (i, (q, \gamma))$. Recall that x stands for a copy of $\mathcal{P}$ in state q with pushdown store content γ, reading the i^{th} letter in a^ω. We associate with x a node $x' \in T_r^A$ labeled by $r^A(x') = (\gamma\Downarrow_1, q)$. Recall that x' stands for a copy of $\mathcal{A}$ in state q, reading node $\gamma\Downarrow_1 \in T_i^A$. Both nodes are labeled by the same state q of $\mathcal{A}$. The pushdown store content of $\mathcal{P}$ is the location of $\mathcal{A}$ in T_i^A.

We prove by induction that we can build $\langle T_r^A, r^A \rangle$ in such a way. We start from the root $\epsilon \in T_r^{PD}$ labeled by $r^{PD}(\epsilon) = (0, (q_0, (\perp, d_0)))$. The root $\epsilon \in T_r^A$ is labeled by $r^A(\epsilon) = (\epsilon, q_0)$. which serves as the base case for the induction.

Given a node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (i, (q, \gamma))$ where $\gamma = (d, \upsilon)\gamma'$, by the induction assumption x is associated with a node $x' \in T_r^A$ labeled by $r^A(x') = (\gamma\Downarrow_1, q)$.

Suppose that x has d successors $c_1 \cdot x, \ldots, c_d \cdot x$ labeled $r^{PD}(c_i \cdot x) = (i+1, (q_i, \beta_i \cdot \gamma'))$. Then there exists a set $Y = \{(q_1, \beta_1), \ldots, (q_d, \beta_d)\}$ such that $Y \models \delta(q, a, (d, \upsilon))$. It must be the case that the set $Y' = \{(q_1, \Delta_1), \ldots, (q_d, \Delta_d)\}$ where Δ_i is $\uparrow$ if β_i is ϵ, ε if β_i is (υ, d) and υ' if β_i is $(\upsilon', d')(\upsilon, d)$ satisfies $\eta(q, L(d))$. We add $c_1 \cdot x', \ldots, c_d \cdot x'$ to T_r^A as the successors of x', and label them $r^A(c_i \cdot x') = (\Delta_i \cdot \gamma\Downarrow_1, q)$. It is simple to see that $\langle T_r^A, r^A \rangle$ is a valid run tree of $\mathcal{A}$ on T_i^A.

[4]We note that it is sufficient to use the set of states D as store alphabet for $\mathcal{P}$. We do not need the location in the tree but rather its labeling, which can be deduced from the run of $\mathcal{D}$. Including Υ on the store, facilitates the proof.

We have to show now that (T_r^A, r^A) is accepting. Take an infinite path $\pi' \subseteq T_r^A$. The path π' matches a path π in T_r^{PD} that is labeled by the same states. Hence, if π satisfies F, then π' satisfies F'.

Assume now that $\mathcal{A}$ accepts T_i^A. There exists an accepting run $\langle T_r^A, r^A \rangle$ of $\mathcal{A}$ on T_i^A. We construct an accepting run $\langle T_r^{PD}, r^{PD} \rangle$ of $\mathcal{P}$ on a^ω.

We assume by induction that every node $x' \in T_r^A$ labeled by $r^A(x') = (\alpha, q)$ is associated with some node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (i, (q, \alpha \times r_{\mathcal{D}}(\alpha)))$ for some i. As before the root ϵ of T_r^A is labeled by $r^A(\epsilon) = (\epsilon, q_0)$. We label the root $\epsilon \in T_r^{PD}$ by $r^{PD}(\epsilon) = (\epsilon, (q_0, (\bot, d_0)))$. This serves as the induction base.

Given some node $x' \in T_r^A$ labeled by $r^A(x') = (\alpha, q)$ where $\alpha = \upsilon\alpha'$, by induction assumption there is a node $x \in T_r^{PD}$ labeled by $r^{PD}(x) = (i, (q, \alpha \times r_{\mathcal{D}}(\alpha)))$ where $\alpha \times r_{\mathcal{D}}(\alpha) = (\upsilon, d)(\alpha' \times r_{\mathcal{D}}(\alpha'))$.

Suppose x' has d successors $c_1 \cdot x', \ldots, c_d \cdot x' \in T_r^A$ labeled by $r^A(c_i \cdot x') = (\alpha_i, q_i)$. Then there exists a set $Y' = \{(\Delta_1, q_1), \ldots, (\Delta_d, q_d)\}$ such that $\alpha_i = \Delta_i \cdot \alpha$ and $Y' \models \eta(q, \tau(\alpha))$. We know that $Y = \{(q_1, \beta_1), \ldots, (q_d, \beta_d)\}$ where β_i is ϵ if Δ_i is $\uparrow$, (υ, d) if Δ_i is ε, and $(\upsilon', \rho(d, \upsilon'))(\upsilon, d)$ if $\Delta_i = \upsilon'$ satisfies $\delta(q, a, (\upsilon, d))$. We add $c_1 \cdot x, \ldots c_d \cdot x$ to T_r^{PD} as the successors of x, and label them $r^{PD}(c_i \cdot x) = (i + 1, (q_i, \beta_i \cdot (\alpha' \times r_{\mathcal{D}}(\alpha'))))$. Obviously, the result is a valid run tree of $\mathcal{P}$ on a^ω.

Every path in T_r^{PD} is associated with a path in T_r^A. We conclude that $\langle T_r^{PD}, r^{PD} \rangle$ satisfies F. □

□

Chapter 5

Micro-Macro Stack Systems: A New Frontier of Elementary Decidability for Sequential Systems

We define the class of micro-macro stack graphs, a new class of graphs modeling infinite-state sequential systems with a decidable model-checking problem. Micro-macro stack graphs are the configuration graphs of stack automata whose states are partitioned into micro and macro states. Nodes of the graph are configurations of the stack automaton where the state is a macro state. Edges of the graph correspond to the sequence of micro steps that the automaton makes between macro states. We prove that this class strictly contains the class of prefix-recognizable graphs. We give a direct automata-theoretic algorithm for model checking μ-calculus and LTL formulas over micro-macro stack graphs.

5.1 Introduction

One of the most significant developments in the area of formal design verification is the discovery of algorithmic methods for verifying on-going behaviors of reactive systems [CE81, QS81, LP85, CES86, VW86]. In *model-checking*, we verify the correctness of a system with respect to a desired behavior by checking whether a mathematical model of the system satisfies a formal specification of this behavior (for a survey, see [CGP99]). Traditionally, model checking is applied to *finite-state* systems, typically modeled by labeled state-transition graphs, and to behaviors that are formally specified as *temporal-logic* formulas or *automata on infinite objects*. Symbolic methods that enable model-checking of very large state spaces, and the great ease of use of fully algorithmic methods, led to industrial acceptance of model-checking [BLM01, CFF$^+$01].

In recent years, researchers extended the applicability of model-checking to infinite-state systems. An active field of research is model-checking of infinite-state *sequential systems*. These are systems in which each state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The origin of this research is the result of Müller and Schupp that the monadic second-order theory of *context-free* graphs is decidable [MS85]. As the complexity involved in that decidability result is nonelementary, researchers sought decidability results of elementary complexity. At the same time, researchers sought decidability results for larger classes of systems. Algorithms for simpler logics and more general systems have been proposed. The most powerful results

are an exponential-time algorithm by Burkart for model checking the μ-calculus with respect to *prefix-recognizable* graphs [Bur97b] and decidability of monadic second-order theory of *high-order pushdown graphs* [KNU03]. See also [BS95, Cau96, Wal96, BE96, BQ96, BEM97, Bur97a, FWW97, BS99, BCMS00, KV00]

The class of high-order pushdown graphs strictly contains the class of prefix-recognizable graphs [KNU03], and the class of prefix-recognizable graphs strictly contains the class of *pushdown* graphs [Cau96], which in turn strictly contains the class of context-free graphs [CM90]. These classes are defined in terms of certain rewrite rules. More powerful notion of rewrite rules yield even larger classes of graphs. The class of *synchronized* rational graphs strictly contains the class of high order pushdown graphs and in turn is strictly contained in the class of *rational* graphs. Only the first-order theory of synchronized rational graphs is, however, decidable (cf. [Büc60, Tho01]). It is undecidable even to determine if some vertex is reachable from another vertex (cf. [Tho01]). For rational graphs even first-order theory is undecidable [Mor00]. To the best of our knowledge the class of prefix-recognizable graphs is the largest class of graphs modeling sequential systems for which there is an elementary model checking algorithm of μ-calculus.

In this paper we present the class of *micro-macro stack graphs*, which strictly contains the class of prefix-recognizable graphs and for which model checking μ-calculus and LTL formulas is decidable in elementary time. Every graph in our class has a simple finite representation in terms of natural rewrite rules. The extension from prefix-recognizable graphs to micro-macro stack graphs is analogous to the extension from pushdown graphs to prefix-recognizable graphs, as we now explain.

The nodes of a pushdown graph are words over some finite alphabet. Such a word represents the store content and the internal state of a pushdown automaton. A transition corresponds to the move of a pushdown automaton when reading some input letter (i.e. popping the top of the store and pushing a finite sequence). The nodes of prefix-recognizable graphs are again words representing the internal state with the content of the pushdown store. Every transition is a triplet of regular languages. Application of the rewrite rule $\langle \alpha, \beta, \gamma \rangle$ on a node x consists of finding a partition zz' of x such that z is in the regular language α and z' is in the regular language β, and then replacing the prefix z by some prefix y in the regular language γ, reaching node yz'. Prefix-recognizable graphs correspond to the configuration graphs of pushdown automata when the ϵ-transitions are factored out [Sti00, Blu01]. Indeed, a pushdown automaton can do a series of ϵ-transitions that remove z from the pushdown store while checking that z is in the language α. Making sure that the suffix z' is in the language β can be done by adding information to the store. Finally, another sequence of ϵ-transitions adds y to the store. We can think of the ϵ-transitions as *micro* steps that are not exhibited in the prefix-recognizable graph. Then, advancing transitions of the pushdown automaton are *macro* steps that are exhibited in the prefix-recognizable graph.

There is another way to let the pushdown automaton check that the suffix z' is in the language β. This is by allowing the automaton to read the entire contents of the store. This is the type of behavior of *stack* automata [GGH67b, GGH67a, HU79]. Just like pushdown automata, stack automata have a finite but unbounded store, they can change only the top of the store by either removing the letter on top of the store or by adding a finite sequence of letters on top of the store. Unlike pushdown automata, stack automata can read the entire contents of their store. A stack automaton can navigate on its store, checking its entire contents. It can change the contents of the store only when it visits the top of the store. Thus a prefix-recognizable graph can be viewed also as the pruned configuration graph of a stack automaton. The following sequences of micro steps are removed: (a) removing the prefix of the pushdown store while checking that it is in the regular

language α, (b) going to the bottom of the store and checking that the remaining suffix is in the language β, and (c) adding a sequence of letters from the regular language γ to the store.

In our framework we offer a more flexible partition into micro and macro states. We let the automaton itself designate which states should be left unnoticed and which states correspond to a change in the graph. We refer to such stack automata whose state set is partitioned into micro and macro states as micro-macro stack automata. The nodes of a micro-macro stack graph correspond to configurations of a micro-macro stack automaton whose state is a macro state. Edges of the graph correspond to the change performed via a sequence of micro states.

We show that the class of micro-macro stack graphs strictly contains the class of prefix-recognizable graphs. We give two examples of micro-macro stack graphs that are not prefix-recognizable. First, as mentioned, prefix-recognizable graphs are the configuration graphs of pushdown automata. Thus, the prefix-recognizable graphs, when considered as acceptors of languages, recognize the context-free languages [Sti00]. We give a micro-macro stack system whose graph accepts a language that is not context free. Second, under some conditions on prefix-recognizable graphs, Blumensath gives information on the size of the encoding of the nodes of a prefix-recognizable graph [Blu01]. We give a micro-macro stack system whose graph does not meet Blumensath' characterization. The class of micro-macro stack graphs is (strictly) contained in the class of synchronized rational graphs. Indeed, we show that model checking the μ-calculus over micro-macro stack graphs is decidable in elementary time. Thus, micro-macro stack graphs constitute a new frontier of elementary decidability for sequential systems.

Our model checking algorithms are automata based. The automata-theoretic approach to verification uses the theory of automata as a unifying paradigm for program specification, verification, and synthesis [VW94, Kur94, KVW00]. Automata enables the separation of the logical and the algorithmic aspects of reasoning about systems, yielding clean and in many cases asymptotically optimal algorithms. The automata-theoretic framework for reasoning about finite-state systems has proven to be very versatile. Recently, the automata-theoretic approach to verification has been extended to infinite-state sequential systems [KV00, KPV02a]. Our model-checking algorithms for micro-macro stack graphs extend the algorithms in [KV00, KPV02a]. In general, the automata-theoretic approach to branching-time model checking uses a reduction to the emptiness problem of alternating tree automata. The automata-theoretic approach to linear-time model checking uses a reduction to the emptiness problem of nondeterministic word automata. In the following we show that for micro-macro stack graphs, the model checking of branching-time specifications (linear-time, respectively) can be reduced to the emptiness problem of alternating stack tree automata. (nondeterministic stack word automata, respectively).

The class of micro-macro stack graphs is contained in the class of high order pushdown graphs. In order to check the contents of the stack a high order pushdown puts a fresh copy of the first order pushdown on the second order pushdown and removes the top of the first order pushdown. The information is not lost, it is stored in the old copy in the second order pushdown. It follows that the monadic second-order theory of micro-macro stack graphs is decidable.

5.2 Transition Graphs and Rewrite Systems

A *labeled transition graph* is $G = \langle \Sigma, S, L, \rho, s_0 \rangle$, where Σ is a finite set of labels, S is a (possibly infinite) set of states, $L : S \to \Sigma$ is a labeling function, $\rho \subseteq S \times S$ is a transition relation, and $s_0 \in S$ is an initial state. When $\rho(s, s')$, we say that s' is a *successor* of s, and s is a *predecessor*

of s'. For a state $s \in S$, we denote by $G^s = \langle \Sigma, S, L, \rho, s \rangle$, the graph G with s as its initial state. An *s-computation* is an infinite sequence $s_0, s_1, \ldots \in S^\omega$ such that $s_0 = s$ and for all $i \geq 0$, we have $\rho(s_i, s_{i+1})$. An s-computation $s_0, s_1, \ldots$ induces the *s-trace* $L(s_0) \cdot L(s_1) \cdots$. The set $\mathcal{T}_s$ is the set of all s-traces and $\mathcal{T}_G = \mathcal{T}_{s_0}$ is the set of all *initialized* traces in G.

A *rewrite system* is $R = \langle \Sigma, V, Q, L, T, q_0, \top, \bot \rangle$, where Σ is a finite set of labels, V is a finite alphabet, Q is a finite set of states, $L : Q \times V^+ \to \Sigma$ is a labeling function, T is a finite set of rewrite rules, to be defined below, $q_0 \in Q$ is an initial state, $\top \in V$ is a store-top symbol, and $\bot \in V$ is a store-bottom symbol. We assume that the store-top symbol is moved whenever the store is extended. We assume that both store-bottom and store-top cannot be removed from nor added to the store and that the system does not try to go below the store-bottom or above the store-top. The set of *configurations* of the system is a subset of $Q \times V^+$. Intuitively, the system has finitely many control states and an unbounded store. Thus, in a configuration $(q, x) \in Q \times V^+$ we refer to q as the *control state* and to x as the *store*.

We consider here the well known *prefix-recognizable* systems and introduce *micro-macro stack systems*. In a *prefix-recognizable* system, each rewrite rule is $\langle q, \alpha, \beta, \gamma, q' \rangle \in Q \times reg(V) \times reg(V) \times reg(V) \times Q$, where $reg(V)$ is the set of regular expressions over V. Thus, $T \subseteq Q \times reg(V) \times reg(V) \times reg(V) \times Q$. For a word $w \in V^*$ and a regular expression $r \in reg(V)$ we write $w \in r$ to denote that w is in the language of the regular expression r. We note that the standard definition of prefix-recognizable systems does not include control states [Cau96] (nor store-bottom or store-top). Indeed, a prefix-recognizable system without states can simulate a prefix-recognizable system with states by having the state as the first letter of the unbounded store. We use prefix-recognizable systems with control states for the sake of uniform notation.

In a *micro-macro stack* system (or *mMs* system for short), the set V contains a special symbol $\uparrow$, the set of states Q is partitioned into the set m of micro states and the set M of macro states, and every configuration contains exactly one occurrence of the symbol $\uparrow$. More formally, denote $V' = V \setminus \{\uparrow, \top, \bot\}$ then the set of possible store contents is $STORES = \{\top\} \cdot (V')^* \cdot \{\uparrow\} \cdot (V')^* \cdot \{\bot\} \cup \{\uparrow\} \cdot \{\top\} \cdot (V')^* \cdot \{\bot\}$ (recall that $\bot$ and $\top$ cannot be removed from nor added to the store). Each rewrite rule of an mMs system is $\langle q, A, act, q' \rangle \in Q \times (V \setminus \{\uparrow\}) \times ACT \times Q$ where $ACT \subseteq \{pop, md, mu, sp\} \cup \{push(z) \mid z \in (V')\}$. Here, *md*, *mu*, and *sp* stand for *move down*, *move up*, and *stay put*, respectively. A configuration in $m \times STORES$ is a *micro configuration* and a configuration in $M \times STORES$ is a *macro configuration*. The labeling function L associates with every configuration of a rewrite system a label. The labeling depends only on one letter of the store x, as we explain below. Thus, we may write $L : Q \times V \to \Sigma$. When a rewrite system R is in configuration (q, x) such that $L(q, x) = a$ for some $a \in \Sigma$ we say that R *signals* a.

A rewrite system R induces a labeled transition graph $G_R = \langle \Sigma, Q \times V^+, L', \rho_R, (q_0, x_0) \rangle$. A labeled transition graph that is induced by a prefix-recognizable system is called a *prefix-recognizable graph*. A labeled transition graph that is induced by an mMs system is called a *micro-macro stack graph* (or *mMs graph* for short). For a prefix-recognizable system the states of G_R are the configurations of R, the initial store content is $x_0 = \top\bot$, and $\langle (q, z), (q', z') \rangle \in \rho_R$ if there is a rewrite rule in T leading from configuration (q, z) to configuration (q', z'). For an mMs system the states of G_R are the macro configurations of R, the initial store content is $x_0 = \top\uparrow\bot$, and $\langle (q, z), (q', z') \rangle \in \rho_R$ if there is a sequence of rewrite rules in T leading from configuration (q, z) to configuration (q', z') by a series of micro configurations. Formally, if R is a prefix-recognizable system, then $\rho_R((q, x \cdot y), (q', x' \cdot y))$ if there are regular languages α, β, and γ such that $x \in \alpha$, $y \in \beta$, $x' \in \gamma$, and $\langle q, \alpha, \beta, \gamma, q' \rangle \in T$. The labeling of a state (q, Az) is $L(q, A)$. In order to

consider the case of an mMs system we need a few definitions. Let $\mathcal{H}(w_2\uparrow z w_1\bot) = z$, where $z \in V$, $\mathcal{H}(\uparrow\top w\bot) = \top$, and $\mathcal{H}(w\uparrow\bot) = \bot$. This describes the letter in the store read by the mMs system. In order to define the effect of applying a rewrite rule on a configuration we define the partial function $\mathcal{B} : STORES \times ACT \to STORES$. This function gives the new content of the store and is defined below. We assume that the system never moves up when it is at the top of the store nor down when it is at the bottom of the store.

- $\mathcal{B}(\uparrow\top z w\bot, pop) = \uparrow\top w\bot$.
- $\mathcal{B}(\uparrow\top w\bot, push(z)) = \uparrow\top z w\bot$.
- $\mathcal{B}(w_2\uparrow z w_1\bot, md) = w_2 z\uparrow w_1\bot$.
- $\mathcal{B}(w_2 z\uparrow w_1\bot, mu) = w_2\uparrow z w_1\bot$.
- $\mathcal{B}(s, sp) = s$.

A sequence of micro steps from a macro configuration (q, z) to macro configuration (q', z') is a sequence (q_1, z_1), (q_2, z_2), ..., (q_n, z_n) such that $(q, z) = (q_1, z_1)$, $(q', z') = (q_n, z_n)$, forall $1 < i < n$ we have $q_i \in m$, and forall $1 \leq i < n$ there exists a rewrite rule $\langle q_i, \mathcal{H}(z_i), act, q_{i+1}\rangle \in T$ and $\mathcal{B}(z_i, act) = z_{i+1}$. Finally, for a pair of macro configurations we have $\langle (q, z), (q', z')\rangle \in \rho_R$ if there exists a sequence of micro steps from (q, z) to (q', z'). Note that the mMs system collects information on the content of the stack and stores it in its control state. In particular, when standing on top of the store the mMs system always reads the symbol $\top$ and relies solely on the control state to choose its next action. The labeling of a state (q, z) is $L(q, \mathcal{H}(z))$.[1] We demand that the initial state of an mMs system be a macro state. This way we are ensured that the induced mMs graph has a single initial state. The work in this paper can be easily generalized for the case that there are many initial states.

Example 5.2.1 *Consider the following mMs* $R = \langle \{a, b, c, d\}, \{A, \top, \bot\}, Q, L, T, q_0, \top, \bot\rangle$ *where* $Q = \{q_0, q_g, q_a, q_b, q_c, q_d\}$, *the state* q_g *is the only micro state,* $L(q_\gamma, A) = L(q_\gamma, \top) = L(q_\gamma, \bot) = \gamma$ *for* $\gamma \in \{a, b, c, d\}$ *and* $L(q_0, A) = L(q_0, \bot) = d$. *Finally,* T *includes the following transitions.*

- $\langle q_0, \bot, mu, q_g\rangle$ - *signal d (state* q_0*) and move to a guessing state.*
- $\langle q_g, \top, push(A), q_g\rangle$ - *guess a number of A and put it on the store.*
- $\langle q_g, \top, md, q_a\rangle$ - *nondeterministically decide to start signaling a.*
- $\langle q_a, A, md, q_a\rangle$ - *go down the store while reading A and signaling a.*
- $\langle q_a, \bot, mu, q_b\rangle$ - *when reaching the bottom of the store start signaling b.*
- $\langle q_b, A, mu, q_b\rangle$ - *go up the store while reading A and signaling b.*
- $\langle q_b, \top, md, q_c\rangle$ - *when reaching the top of the store start signaling c.*

[1] The choice to set the labeling according to $\mathcal{H}(z)$ is the most general. We can emulate labeling according to the top of the store by remembering the top of the store as part of the control. Similarly, regular labeling, that depends on the membership of the entire store content in some regular language can be emulated by remembering the state of a deterministic finite automaton for the regular language on the store.

- $\langle q_c, A, md, q_c \rangle$ - *go down the store while reading* A *and signaling* c.
- $\langle q_c, \perp, sp, q_d \rangle$ - *when reaching the bottom of the store start signaling* d.
- $\langle q_d, \perp, sp, q_d \rangle$ - *signal* d *indefinitely.*

This system produces a 'star' of infinite degree. The center of this star is the initial state and each 'ray' is a sequence $da^n b^n c^n d^\omega$ *for some* n. *It does so by guessing some number of* A *and put them on the store. Then it moves up and down the store while signaling* a, b, *and* c.

Example 5.2.2 *Consider the mMs system* $R = \langle \{a, b\}, \{\top, \perp, A\}, \{q_0, r, l, p\}, L, T, q_0, \top, \perp \rangle$ *where* $m = \{r\}$, $M = \{q_0, l, p\}$. *The labeling function* L *associates with state* l *the symbol* a *and with states* q_0 *and* p *the symbol* b. *The set of rewrite rules* T *contains the following rules.*

- $\langle q_0, \perp, sp, r \rangle$ - *move to micro state* r.
- $\langle r, \perp, mu, l \rangle$ - *bottom of the store, switch to state* l.
- $\langle l, \top, sp, p \rangle$ - *top of the store, switch to state 'push'.*
- $\langle l, A, mu, l \rangle$ - *move up the store.*
- $\langle p, \top, push(A), r \rangle$ - *extend the store, switch to state* r.
- $\langle r, \top, md, r \rangle, \langle r, A, md, r \rangle$ - *move down.*

This system is in fact a unary counter. It 'outputs' 1, then 2, then 3, and so on ad-infinitum. It does so by going to the bottom of the store, then while it goes to the top of the store it signals a *with every move. When reaching the top of the store it extends the store with one symbol and signals one* b. *Thus, the first two* b *symbols are separated by one* a. *Then before the third* b *symbol there are 2* a*s, and generally the* i*th* b *is separated from the next* b *by the sequence* a^i.

It is quite easy to see that given a prefix-recognizable system R we can construct an mMs system R' such that G_R and $G_{R'}$ are isomorphic. The set of macro states of R' correspond to the set of states of R and the set of micro states of R' correspond to the states of automata that recognize the regular languages in the transitions of R. The mMs system R' mimics a transition $\langle q, \alpha, \beta, \gamma, q' \rangle$ of R by removing a sequence of letters from the store while simulating a run of the automaton for the regular language α on the removed sequence. Then the mMs system goes to the bottom of the stack and checks that what is left on the stack is a word in the regular language β. Finally, the mMs system guesses a word in γ (letter by letter) and adds it to the store[2]. In the sequel we show that the opposite is not true. There are mMs graphs with no isomorphic prefix-recognizable graph. The fact that an mMs system remembers in addition to the stack contents a location in the stack is enough to give it extra power over prefix-recognizable systems.

[2] A similar construction appears in Section 5.3.1.

5.3 Non Prefix-Recognizable mMs graphs

We give two examples of mMs graphs for which there exist no isomorphic prefix-recognizable graphs[3]. We use two methods to prove that these graphs are not prefix-recognizable. First, we use simple language considerations. We show that our graph, if represented by a prefix-recognizable system, induces a pushdown automaton over finite words that accepts a language that is not context-free. Second, we use a characterization of prefix-recognizable graphs by Blumensath [Blu01]. We give another graph and prove that it does not satisfy the requirements of Blumensath.

We define isomorphism with respect to two graphs $G = \langle \Sigma, S, L, \rho, s_0 \rangle$ and $G' = \langle \Sigma, S', L', \rho', s'_0 \rangle$, with the same alphabet. A bijection $I : S \to S'$ is an *isomorphism* between G and G' iff forall $s, t \in S$ we have $L(s) = L(I(s))$ and $\rho(s,t)$ iff $\rho'(I(s), I(t))$.

5.3.1 An mMs Graph Recognizing a non Context-Free Language

We use the well known non context-free language $\{a^n b^n c^n \mid n \in \mathcal{N}\}$ to prove that a graph is not prefix-recognizable. In fact, for many languages, if L is not context free but it can be recognized by a finite stack automaton, we can construct an mMs graph G_L whose 'language' is L. The graph G_L should be another example of an mMs graph that has no isomorphic prefix-recognizable graph.

We first need some definitions. A *pushdown automaton over finite words* (or *PD-NFW* for short) is $P = \langle \Sigma, \Gamma, Q, \rho, q_0, \bot, F \rangle$ where Σ is a finite input alphabet, Γ is a finite pushdown alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, $\bot \in \Gamma$ is a store-bottom symbol, and $F \subseteq Q$ is a set of accepting states. The transition function $\rho : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ associates with every state $q \in Q$, input letter $\sigma \in (\Sigma \cup \{\epsilon\})$, and pushdown letter $A \in \Gamma$ a finite set of possible transitions $\rho(q, \sigma, A)$.

A *configuration* of a PD-NFW is a pair $(q, w) \in Q \times \Gamma^*$ where q is the state of the automaton and w is the content of the pushdown store. A *run* of a PD-NFW over a finite word $w \in \Sigma^*$ is a sequence of configurations and locations $r \in (Q \times \Gamma^* \times \{0, \ldots, |w|+1\})^*$ such that $r_0 = \langle (q_0, \bot), 0 \rangle$ and for every $0 \leq i < |r|$ we have $r_i = \langle (q, A \cdot x), j \rangle$, $r_{i+1} = \langle (q', y \cdot x), j + \Delta \rangle$ and either $\langle q', y \rangle \in \rho(q, w_j, A)$ and $\Delta = 1$, or $\langle q', y \rangle \in \rho(q, \epsilon, A)$ and $\Delta = 0$. A transition from $\langle (q, x), j \rangle$ to $\langle (q', x'), j \rangle$ is called an *ϵ-transition*. Otherwise it is an *advancing transition*. A run is accepting if $r_{|r|} = \langle f, x, |w|+1 \rangle$ for some state $f \in F$ and forall $j < |r|$ we have that $r_j = \langle s, y, k \rangle$ such that $k \leq |w|$. A word w is *accepted* by P if there exists an accepting run of P on w. The *language* $\mathcal{L}(P)$ of P is the set of words accepted by P.

An automaton is an NFW if $\Gamma = \{\bot\}$ and the transition function is restricted to $\rho : Q \times \Sigma \times \{\bot\} \to 2^{Q \times \{\bot\}}$. In this case we write $N = \langle \Sigma, Q, \rho, q_0, F \rangle$ and $\rho : Q \times \Sigma \to 2^Q$. In case that for every letter $\sigma \in \Sigma$ and state $q \in Q$ we have $|\rho(q, \sigma)| = 1$ we say that N is deterministic (DFW).

Consider the mMs system $R = \langle\ \{a, b, c, d\},\ \{A, \top, \bot\},\ Q,\ L,\ T,\ q_0,\ \top,\ \bot\ \rangle$ from Example 5.2.1.

Claim 5.3.1 *There is no prefix-recognizable system S such that G_S is isomorphic to G_R.*

Proof: Given a PD-NFW $P = \langle \Sigma, \Gamma, Q, \rho, q_0, \bot, F \rangle$ we construct the graph G_P where the set of vertices is the set of configurations of P from which there exists an advancing transition. An edge connects (q, w) to (q', w') if there exists an advancing transition followed by a sequence of ϵ

[3] In fact, for our graph there does not exist a *bisimilar* prefix-recognizable graph. Our proof can be extended for the case of bisimilarity.

transitions leading from (q, w) to (q', w'). Finally, we label a configuration (q, Aw) by the set of letters $\sigma \in \Sigma$ such that $\rho(q, \sigma, A) \neq \emptyset$. Formally, let $G_P = \langle 2^\Sigma, S, L, \rho_P, s_0 \rangle$ where $S = \{(q, A \cdot w) \in Q \times \Gamma^+ \mid \exists \sigma \in \Sigma \text{ s.t. } \rho(q, \sigma, A) \neq \emptyset\}$, $s_0 = (q_0, \bot)$, and $L(q, Aw) = \{\sigma \mid \rho(q, \sigma, A) \neq \emptyset\}$. We have $((q, Aw), (q', w')) \in \rho_P$ if there exists a run $(q, Aw, 0), (q_1, w_1, 1), \ldots, (q_n, w_n, 1), (q', w', 1)$ on a word $\sigma \in L(q, Aw)$ (note that $|\sigma| = 1$ and that this run is not necessarily accepting).

It is known that the graph of a prefix-recognizable system is isomorphic to the configuration graph of a PD-NFW [Sti00, Blu01]. We extend this result by showing that given a prefix-recognizable system $R_{pr} = \langle \Sigma_{pr}, V_{pr}, Q_{pr}, L_{pr}, T_{pr}, q_0^{pr}, \top, \bot \rangle$ we can construct a PD-NFW $P = \langle \Sigma, \Gamma, Q_{pd}, \rho, q_0^{pd}, \underline{\bot}, F \rangle$ such that every state s of G_P has $|L(s)| = 1$ and if we restrict our attention to states reachable from $(q_0^{pd}, \underline{\bot})$ in G_P and from $(q_0^{pr}, \top\bot)$ in $G_{R_{pr}}$ the two are isomorphic.

We first need a few definitions. Consider the prefix-recognizable system $R_{pr} = \langle\ \Sigma_{pr},\ V_{pr},\ Q_{pr},\ L_{pr},\ T_{pr},\ q_0^{pr},\ \top,\ \bot\ \rangle$. For a rewrite rule $t_i = \langle\ q,\ \alpha_i,\ \beta_i,\ \gamma_i,\ q'\ \rangle \in T_{pr}$, let $\mathcal{U}_\lambda = \langle V_{pr}, Q_\lambda, \eta_\lambda, q_\lambda^0, F_\lambda \rangle$, for $\lambda \in \{\alpha_i, \beta_i, \gamma_i\}$, be the NFW for the language of the regular expression λ. We assume that all initial states have no incoming edges and that all accepting states have no outgoing edges. We collect all the states of all the automata for α, β, and γ regular expressions. Formally, $Q_\alpha = \bigcup_{t_i \in T} Q_{\alpha_i}$, $Q_\beta = \bigcup_{t_i \in T} Q_{\beta_i}$, and $Q_\gamma = \bigcup_{t_i \in T} Q_{\gamma_i}$. Let $\{\beta_1, \ldots, \beta_n\}$ be the set of regular expressions β_i such that there is a rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \in T_{pr}$. Let $\mathcal{D}_{\beta_i} = \langle V_{pr}, D_{\beta_i}, \eta_{\beta_i}, q_{\beta_i}^0, F_{\beta_i} \rangle$ be the deterministic automaton for the **reverse** language of β_i. For a word $x \in V_{pr}^*$, we denote by $\eta_{\beta_i}(x)$ the unique state that $\mathcal{D}_{\beta_i}$ reaches after reading the word x. Let $\Upsilon = V_{pr} \times \Pi_{1 \leq i \leq n} D_{\beta_i}$. For a letter $\upsilon \in \Upsilon$, let $\upsilon[i]$, for $i \in \{0, \ldots n\}$, denote the i-th element in υ (that is, $\upsilon[0] \in V_{pr}$ and $\upsilon[i] \in D_{\beta_i}$ for $i > 0$). Let $\eta_\mathcal{D} : (\Upsilon \times V) \to \Upsilon$ denote the effect of reading a letter V on the states of the automata $\mathcal{D}_{\beta_i}$. Formally, $\eta_\mathcal{D}(\upsilon, A) = \langle A, \eta_{\beta_1}(\upsilon[1], A), \ldots, \eta_{\beta_n}(\upsilon[n], A) \rangle$. We use $\underline{\bot} = \langle \bot, q_0^{\beta_1}, \ldots, q_0^{\beta_n} \rangle$ as store-bottom symbol of the PD-NFW. Given a word $w = w_0, \ldots, w_m \in V_{pr}^* \cdot \bot$ we denote by $r^{\beta_i}(w) = r_{\beta_i}^0, \ldots, r_{\beta_i}^m$ the unique run of $\mathcal{D}_{\beta_i}$ on the reverse of w. We denote by $w \times r^{\beta_1}(w) \times \cdots \times r^{\beta_n}(w)$ the word $\upsilon_0, \upsilon_1, \ldots, \upsilon_m$ such that forall $0 \leq j < m$ we have $\upsilon_j[0] = w_j$ and $\upsilon_m[0] = \bot$ and forall $0 \leq j \leq m$ and $1 \leq i \leq n$ we have $\upsilon_j[i] = r_{\beta_i}^{m-j}$.

Consider the PD-NFW $P = \langle \Sigma_{pr}, \Upsilon, Q_{pr} \cup (T_{pr} \times (Q_\alpha \cup Q_\gamma)), \rho, q_0^{pr}, \underline{\bot}, \emptyset \rangle$ where the transition function is defined as follows.

- For a state $q \in Q_{pr}$ and $\upsilon \in \Upsilon$ we have

$$\rho(q, \sigma, \upsilon) = \begin{cases} \{\langle (t_i, s), \upsilon \rangle \mid t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \text{ and } s = q_{\alpha_i}^0\} & \text{If } \sigma = L_{pr}(q, \upsilon[0]) \\ \emptyset & \text{If } \sigma \neq L_{pr}(q, \upsilon[0]) \end{cases}$$

- For a state $(t_i, s) \in T_{pr} \times Q_\alpha$ where $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ we have

$$\rho((t_i, s), \sigma, \upsilon) = \begin{cases} \bigcup \begin{array}{l} \{\langle (t_i, s'), \epsilon \rangle \mid s' \in \eta_{\alpha_i}(s, \upsilon[0])\} \\ \{\langle (t_i, s'), \upsilon \rangle \mid s \in F_{\alpha_i}, s' \in F_{\gamma_i}, \text{ and } \upsilon[i] \in F_{\beta_i}\} \end{array} & \text{If } \sigma = \epsilon \\ \emptyset & \text{If } \sigma \in \Sigma_{pr} \end{cases}$$

- For a state $(t_i, s) \in T_{pr} \times Q_\gamma$ where $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ we have

$$\rho((t_i, s), \sigma, \upsilon) = \begin{cases} \bigcup \begin{array}{l} \{\langle (t_i, s'), \upsilon'\upsilon \rangle \mid s \in \eta_{\gamma_i}(s', \upsilon'[0]) \text{ and } \upsilon' = \eta_\mathcal{D}(\upsilon, \upsilon'[0])\} \\ \{\langle q', \upsilon \rangle \mid s = q_{\gamma_i}^0\} \end{array} & \text{If } \sigma = \epsilon \\ \emptyset & \text{If } \sigma \in \Sigma_{pr} \end{cases}$$

We show that the subgraphs of G_P and $G_{R_{pr}}$ that are reachable from the respective initial states are isomorphic. Consider the function $I : Q_{pr} \times V^* \to Q_{pd} \times \Upsilon^*$ where $I(q, x) = (q, x \times r^{\beta_1}(x) \times \cdots \times r^{\beta_n}(x))$.

For every state $q \in Q_{pr}$ the only letter $\sigma \in \Sigma_{pr}$ for which $\rho(q, \sigma, v)$ is defined is $L(q, v[0])$, hence $L_{pr}(q, w) = L_{G_P}(I(q, w))$. For every state $(t, q) \in T_{pr} \times (Q_\alpha \cup Q_\gamma)$ we have $\delta(q, \sigma, v) = \emptyset$ for $\sigma \neq \epsilon$. By definition it is clear that I is an injection. We show that I is a surjection. Notice that, the only way to extend the store is by a transition $\langle (t, s'), v'v \rangle$. However, we demand that $v' = \eta_{\mathcal{D}}(v, v'[0])$. Hence, the only states reachable from $(q_0^{pd}, \underline{\perp})$ are states in the range of I. We show by induction on the distance from $(q_0^{pd}, \underline{\perp})$ that every configuration (q, w') of P is in the range of I. For $(q_0^{pd}, \underline{\perp})$ the proof is immediate. Assume that all configurations whose distance is k are in the range of I. Consider a state (q, w) of G_P of distance $k+1$ from $(q_0^{pd}, \underline{\perp})$. By induction there exists a reachable state (q', w') of $G_{R_{pr}}$ such that $(I(q', w'), (q, w)) \in \rho_P$. Denote $s = I(q', w')$ and $s' = (q, w)$. Then there is a sequence s, $\langle (t, q_\alpha^0), w_0 \rangle$, $\langle (t, q_\alpha^1), w_1 \rangle$, ..., $\langle (t, q_\alpha^m), w_m \rangle$, $\langle (t, q_\gamma^{m'}), x_{m'} \rangle$, ..., $\langle (t, q_\gamma^0), x_0 \rangle, s'$ such that w_0 is the store content of s, $x_0 = w$ is the store content of s', $w_m = x_{m'}$ is a common suffix of w_m and x_0 such that $w_m \in \beta$ (note that $\mathcal{D}_\beta$ runs from the end of w_m to the start of w_m and recognizes words whose reverse is in β). and the runs $q_\alpha^0, \ldots, q_\alpha^m$ and $q_\gamma^0, \ldots, q_\gamma^{m'}$ are witnesses that the prefixes of w_0 and x_0 are in α and γ respectively. We conclude that the state $(q', w[0])$ is reachable in $G_{R_{pr}}$ and $I(q', w[0]) = s'$. In a similar manner we show that $(s, s') \in \rho_{T_{pr}}$ iff $(I(s), I(s')) \in \rho_P$.

Assume by contradiction that there exists a prefix-recognizable system whose graph is isomorphic to G_R. Let $R_{pr} = \langle \{a, b, c, d\}, V, Q, L, T, q_0, \top, \perp \rangle$ be the prefix-recognizable system that produces this graph. Let $P = \langle \{a, b, c, d\}, \Upsilon, Q', \rho, q_0', \underline{\perp}, \emptyset \rangle$ be the PD-NFW such that G_P is isomorphic to $G_{R_{pr}}$. Consider the PD-NFW $P' = \langle \{a, b, c, d\}, \Upsilon, Q' \cup \{acc\}, \rho', q_0', \underline{\perp}, \{acc\} \rangle$, where for every state $q \in Q'$ and letter $A \in \Upsilon$ such that $\rho(q, d, A) \neq \emptyset$ we define $\rho'(q, d, A) = \rho(q, d, A) \cup \{\langle acc, A \rangle\}$ and $\rho'(q, \gamma, A) = \rho(q, \gamma, A)$ for $\gamma \neq d$ or when $\rho(q, d, A) = \emptyset$. Thus, whenever P' reads the letter d it can enter an accepting state. Is is simple to see that the language of P' is $\{d\} \cdot \{a^n b^n c^n \mid n \geq 1\} \cdot \{d\}^+$. However, $\{a^n b^n c^n \mid n \in \mathcal{N}\}$ is not a context-free language [HU79]. contradiction. □

5.3.2 An mMs Graph Violating Prefix-Recognizable Representation Characterization

We give another example of an mMs graph that is not prefix-recognizable. This time, we use characterization of prefix-recognizable graphs by Blumensath [Blu01] to show that the graph is not prefix-recognizable.

We first need some definitions. Let $\mathcal{T}_2 = \{0, 1\}^*$ denote the full binary tree and let $succ_0(x, y)$ and $succ_1(x, y)$ denote the successor relations (that is, for $x, y \in \{0, 1\}^*$ we have $succ_0(x, y)$ true iff $y = x \cdot 0$, and similarly for $succ_1$). We define Monadic Second Order Logic (MSOL) over $\langle \mathcal{T}_2, succ_0, succ_1 \rangle$ as follows. Let $x, y, \ldots$ denote vertex variables and $X, Y, \ldots$ denote set variables. *Atomic formulas* are $x \in X$ and $succ_i(x, y)$ for $i \in 0, 1$. *MSOL formulas* are the formulas obtained from atomic formulas by closing them under Boolean connectives and existential quantification over vertex or set variables. For a full exposition of MSOL we refer the reader to [Tho90].

A graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$ is *MSOL interpretable* in the structure $\langle \mathcal{T}_2, succ_0, succ_1 \rangle$ if there exist MSOL formulas $\varphi_\rho(x, y)$, φ_{s_0}, and for every $\sigma \in \Sigma$ we have $\varphi_\sigma(x)$ such that the graph G' defined below is isomorphic to G. Let $G' = \langle \Sigma, S', L', \rho', s_0' \rangle$ such that $S' = \{t \in \mathcal{T}_2 \mid \exists x. \varphi_\rho(t, x) \vee \varphi_\rho(x, t)\}$, $S_0 = \{t \in \mathcal{T}_2 \mid \varphi_{s_0}(t)\}$ is a singleton $S_0 = \{s_0'\}$, $\rho = \{(x, y) | \varphi_\rho(x, y)\}$ and finally, for every

$R_{a^*}(x', y') =$

$$\forall X \left[\left(\begin{array}{c} (\forall z.z \in X \rightarrow \varphi_a(z)) \wedge \\ (\forall z, z'.z \in X \wedge \varphi_\rho(z, z') \wedge \varphi_a(z') \rightarrow z' \in X) \end{array} \right) \rightarrow (x' \in X \rightarrow y' \in X) \right] \wedge \varphi_a(x') \wedge \varphi_a(y')$$

$$R_{ba^*b}(x, y) = \varphi_b(x) \wedge \varphi_b(y) \wedge \left[\varphi_\rho(x, y) \vee \exists x', y' \left(R_{a^*}(x', y') \wedge \varphi_\rho(x, x') \wedge \varphi_\rho(y', y) \right) \right]$$

Figure 5.1: Interpretation in $\mathcal{T}_2$

state $s \in S'$ we have $l(s) = \{\sigma \mid \varphi_\sigma(s)\}$ is a singleton and $l(s) = \{L(s)\}$. We overload notation and refer to $\langle \varphi_\rho, \varphi_{s_0}, (\varphi_\sigma)_{\sigma \in \Sigma} \rangle$ both as the interpretation in $\mathcal{T}_2$ and the graph that it induces.

Theorem 5.3.2 [Bar97, Blu01] *A graph G is prefix-recognizable iff it is MSOL interpretable in $\langle \mathcal{T}_2, succ_0, succ_1 \rangle$.*

Consider a prefix-recognizable graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$ and a state $s \in S$. Let $Pre^*(s)$ denote the set of states from which s is reachable. Formally, for $S' \subseteq S$ we define $Pre_0(S') = S'$, $Pre_{i+1}(S') = \{t \mid \exists t' \in Pre_i(S') \text{ and } (t, t') \in \rho\}$, and $Pre^*(S') = \bigcup_{i \geq 0} Pre_i(S')$. If $S' = \{s\}$ is a singleton we write $Pre^*(s)$. We define a relation on the states of G, forall s, t we have $s \leq t$ iff $s \in Pre^*(t)$. Let $\langle \varphi_\rho, \varphi_{s_0}, (\varphi_\sigma)_{\sigma \in \Sigma} \rangle$ be some interpretation of G in $\mathcal{T}_2$ and let $I : S \rightarrow \mathcal{T}_2$ be the bijection associating G and $\langle \varphi_\rho, \varphi_{s_0}, (\varphi_\sigma)_{\sigma \in \Sigma} \rangle$. For a state $s \in S$, let $|I(s)|$ denote the length of $I(s) \in \mathcal{T}_2$.

Lemma 5.3.3 [Blu01] *Let G be a prefix-recognizable graph. If the relation $s \leq t$ is a well order on the states of G, then for every interpretation of G in $\mathcal{T}_2$ with bijection I we have $|I(a_n)| = \Theta(n)$ where a_n is the nth element in the ordering.*

Consider the mMs system $R = \langle \{a, b\}, \{\top, \bot, A\}, \{q_0, r, l, p\}, L, T, q_0, \top, \bot \rangle$ from Example 5.2.2.

Theorem 5.3.4 *The system R above is not a prefix-recognizable system.*

Proof: Lemma 5.3.3 gives information on the size of the encoding of states of a well ordered prefix-recognizable graph. The graph above is clearly well ordered. We show a subgraph that is also well ordered. Both the graph and its subgraph are embedded in $\mathcal{T}_2$ and share the same encoding of the states. In each graph we get information on the size of the encoding and this information is contradictory.

Suppose by contradiction that there exists a prefix-recognizable system inducing a graph isomorphic to G_R. Let $G = \langle \Sigma, S, L, \rho, s_0 \rangle$ be the graph of this prefix-recognizable system and let $\langle \varphi_\rho, \varphi_{s_0}, \varphi_a, \varphi_b \rangle$ be some interpretation of G in $\mathcal{T}_2$. Let $I : S \rightarrow \mathcal{T}_2$ denote the isomorphism associating the two. As mentioned the system above produces an infinite path of nodes. Let b_i denote the state x such that $|Pre^*(x) \cap \{y \mid L(y) = b\}| = i$.

Denote by $G' = \langle \{b\}, S', L', \rho', s_0 \rangle$ the graph consisting of the states $S' = \{b_i \mid i \geq 1\}$ and the edges $\rho' = \{(b_i, b_{i+1}) \mid i \geq 1\}$. For every state $b_i \in S'$ we have $L'(b_i) = b$ and as $s_0 = b_1$ we use the same initial state in G'. We show that this graph is also interpretable in $\mathcal{T}_2$. Hence, it is

also a prefix-recognizable graph and Lemma 5.3.3 applies for it. Consider the MSOL formulas in Figure 5.1. For states x', y' we have $R_{a^*}(x', y')$ true iff y' is reachable from x' along a path that visits only states labeled a. For states x, y we have $R_{ba^*b}(x, y)$ true iff both x and y are b states and either y is a successor of x or y is reachable from x along a path that visits only states labeled a.

Consider the structure $\langle R_{ba^*b}, \varphi_{s_0}, \mathbf{true}\rangle$. It is isomorphic to the graph G'. Let $I' : S' \to \mathcal{T}_2$ denote the isomorphism between G' and $\langle R_{ba^*b}, \varphi_{s_0}, \mathbf{true}\rangle$ (actually, I' is I restricted to S'). From Theorem 5.3.2 it follows that G' is a prefix-recognizable graph. It is also the case that G' is well ordered. According to Lemma 5.3.3 we have $|I(b_i)| = \Theta(i)$.

As mentioned the graph G consists of a single path. Hence, the relation $\leq$ is a well order on the states of G. The state b_i is the $\frac{i(i+1)}{2}$ state in this order. According to Lemma 5.3.3 we have $|I(b_i)| = \Theta(i^2)$. Contradiction. □

5.4 Model-Checking mMs Systems

We use the automata-theoretic approach to model checking. In the branching time framework, the automata-theoretic approach reduces the model checking problem to the emptiness problem of an alternating tree automaton over 1-letter alphabet. The alternating tree automaton is the combination of the system and the specification [KVW00]. In the linear time framework, the automata theoretic approach reduces the model checking problem into the emptiness problem of a nondeterministic word automaton. The nondeterministic word automaton is the combination of the system and the negation of the specification [VW86]. Similarly, for mMs systems we solve model checking by a reduction to the respective emptiness problem. As our systems have a stack we use stack automata.

In order to give the most general algorithm we assume that the specification is given as an automaton. In the branching time framework the specification is given as an alternating graph automaton (that accepts all possible graphs that satisfy the specification, see below). In the linear time framework the specification is given as a nondeterministic Büchi word automaton (that accepts all traces not satisfying the specification). Algorithms for converting μ-calculus, CTL, CTL*, and LTL and $S1S$ to automata can be found in the literature [Büc62, VW94, JW95, Var98, KVW00].

5.4.1 Definitions

Nondeterministic Automata A *nondeterministic Büchi automaton on infinite words* (or *NBW* for short) is $N = \langle \Sigma, Q, q_0, \eta, F\rangle$, where N is an NFW that is run over infinite words. A *run* of N on an infinite word $w = w_0, w_1, \ldots$ is an infinite sequence $p_0, p_1, \ldots \in Q^\omega$ such that $p_0 = q_0$ and for all $0 \leq j$, we have $p_{j+1} \in \eta(p_j, w_j)$. For a run $r = p_0, p_1, \ldots$, let $inf(r) = \{q \in Q \mid q = p_i \text{ for infinitely many } i\text{'s}\}$ be the set of all states occurring infinitely often in the run. A run r of an NBW is *accepting* if it visits the set F infinitely often, thus $inf(r) \cap F \neq \emptyset$. A word w is *accepted* by N if N has an accepting run on w. The *language* of N, denoted $\mathcal{L}(N)$, is the set of words accepted by N. We use NBW as our linear-time specification language. A linear-time specification N is the NBW characterizing all the **bad** behaviors. If we want to check that a model satisfies a property φ we have to construct the NBW N for $\neg\varphi$. A labeled transition graph G does not satisfy a NBW N, denoted $G \not\models N$ iff $\mathcal{T}_G \cap \mathcal{L}(N) \neq \emptyset$.

We are especially interested in cases where $\Sigma = 2^{AP}$, for some set AP of atomic propositions AP, and in languages $L \subseteq (2^{AP})^\omega$ definable by NBW or formulas of the linear temporal logic LTL

[Pnu77]. For an LTL formula φ, the *language* of φ, denoted $L(\varphi)$, is the set of infinite words that satisfy φ.

Theorem 5.4.1 [VW94] *For every LTL formula φ, there exists an NBW N_φ with $2^{O(|\varphi|)}$ states, such that $L(N_\varphi) = L(\varphi)$.*

Graph Automata Given a finite set Υ of directions, an Υ-*tree* is a set $T \subseteq \Upsilon^*$ such that if $v \cdot x \in T$, where $v \in \Upsilon$ and $x \in \Upsilon^*$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $v \in \Upsilon$ and $x \in T$, the node x is the *parent* of $v \cdot x$ and $v \cdot x$ is a *child* of x. If $z = x \cdot y \in T$ then z is a descendant of y. There is a natural partial order induced on the nodes of the tree, $x \leq y$ iff y is a descendant of x. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. A *path* π of a tree T is an infinite set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $v \in \Upsilon$ such that $v \cdot x \in \pi$. Our definitions here reverse the standard definitions (e.g., when $\Upsilon = \{0, 1\}$, the successors of 0 are 00 and 10, rather than 00 and 01).

Given two finite sets Υ and Σ, a Σ-*labeled* Υ-*tree* is a pair $\langle T, \tau \rangle$ where T is an Υ-tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, \tau \rangle$ a labeled tree.

For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**, and, as usual, $\wedge$ has precedence over $\vee$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning **true** to elements in Y and assigning **false** to elements in $X \setminus Y$ makes θ true.

An alternating automaton on labeled transition graphs (*graph automaton*, for short) [Wil99] is a tuple $\mathcal{A} = \langle \Sigma, Q, \eta, q_0, \alpha \rangle$, where Σ, Q, q_0 are as in PD-NFW, α specifies the acceptance condition, and $\eta : Q \times \Sigma \to \mathcal{B}^+(\{\varepsilon, \Diamond, \Box\} \times Q)$ is the transition function. Intuitively, when $\mathcal{A}$ is in state q and it reads a state s of G, fulfilling an atom $\langle \Diamond, p \rangle$ (or $\Diamond p$, for short) requires $\mathcal{A}$ to send a copy in state p to some successor of s. Similarly, fulfilling an atom $\Box p$ requires $\mathcal{A}$ to send copies in state p to all the successors of s. The atom $\langle \varepsilon, p \rangle$ (or p, for short) requires $\mathcal{A}$ to send a copy in state p to the node s itself. Thus, like symmetric automata [DW99, Wil99], graph automata cannot distinguish between the various successors of a state and treat them in an existential or universal way.

A run of a graph automaton $\mathcal{A}$ on a labeled transition graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$ is a labeled tree in which every node is labeled by an element of $S \times Q$. A node labeled by (s, q), describes a copy of the automaton that is in the state q of $\mathcal{A}$ and reads the state s of G. Formally, a run is a Σ_r-labeled Γ-tree $\langle T_r, r \rangle$, where Γ is some set of directions, $\Sigma_r = S \times Q$, and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (s_0, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (s, q)$ and $\delta(q, L(s)) = \theta$. Then there is a (possibly empty) set $Y \subseteq (\{\varepsilon, \Diamond, \Box\} \times Q)$, such that Y satisfies θ, and for all $\langle c, q' \rangle \in Y$, the following hold:
   - If $c = \varepsilon$, then there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s, q')$.
   - If $c = \Box$, then for every successor s' of s, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.
   - If $c = \Diamond$, then there is a successor s' of s and $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.

A run $\langle T_r, r\rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. We consider here *parity* acceptance conditions [EJ91]. A parity condition $\alpha = \{F_1, F_2, \ldots, F_m\}$ is a partition of Q. The number m of sets is called the *index* of $\mathcal{A}$. Given a run $\langle T_r, r\rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in S \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies the condition F if the minimal i such that $inf(\pi) \cap F_i \neq \emptyset$ is even. A run $\langle T_r, r\rangle$ is accepting if all paths π in T_r are accepting. The graph G is accepted by $\mathcal{A}$ if there is an accepting run on it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all graphs that $\mathcal{A}$ accepts.

We use graph automata as our branching-time specification language. We say that a labeled transition graph G satisfies a graph automaton $\mathcal{A}$, denoted $G \models \mathcal{A}$, if $\mathcal{A}$ accepts G. Graph automata are as expressive as μ-calculus [JW95, Wil99]. In particular, we have the following.

Theorem 5.4.2 *Given a μ-calculus formula ψ, of length n and alternation depth k, we can construct a graph parity automaton $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi)$ is exactly the set of graphs satisfying ψ. The automaton $\mathcal{A}_\psi$ has n states and index k.*

Stack Automata A *stack alternating automaton* on Υ-trees (ST-APT) is $M = \langle \Sigma, V, Q, \delta, q_0, \top, \bot, \alpha\rangle$ where Σ, Q, q_0, and α are as in graph automata, V is the stack alphabet, and $\top$ and $\bot$ are the store-top and store-bottom symbols (that cannot be removed from nor added to the stack). The transition function is $\delta : Q \times \Sigma \times V \to B^+(Q \times ACT \times \Upsilon)$ where ACT is the set of possible actions as defined for mMs systems. We also use the set of stack configurations $STORES$, the function $\mathcal{H}$ and the function $\mathcal{B}$ as defined for mMs systems.

A run of M on a Σ-labeled Υ-tree $\langle \Upsilon^*, \tau\rangle$ is a $\Upsilon^* \times Q \times STORES$-labeled Γ-tree $\langle T, r\rangle$ where Γ is some set of directions and $\langle T, r\rangle$ satisfies the following.

- $\varepsilon \in T$ and $r(\varepsilon) = (\epsilon, q_0, \top\uparrow\bot)$.
- Consider $x \in T$ with $r(x) = (y, q, z)$ and $\delta(q, \tau(y), \mathcal{H}(z)) = \theta$. Then there exists (a possibly empty) set $\{(q_1, act_1, \Delta_1), \ldots, (q_d, act_d, \Delta_d)\} \models \theta$ and x has d successors $x_1, \ldots x_d$ such that $r(x_j) = (\Delta_j y, q_j, \mathcal{B}(z, act_j))$.

A run is accepting if every infinite path in $\langle T, r\rangle$ satisfies the parity acceptance condition. We are interested in in nondeterministic stack automata over infinite words ($|\Upsilon| = 1$) and alternating stack automata with 1-letter input alphabet.

An automaton is nondeterministic if in every transition exactly one copy of the automaton is sent in every direction in Υ. Formally, an automaton is nondeterministic if for every state $q \in Q$, letter $\sigma \in \Sigma$, and letter $A \in V$ there exists some set I such that $\delta(q, \sigma, A) = \bigvee_{i \in I} \bigwedge_{\upsilon \in \Upsilon} (s_{i,\upsilon}, act_{i,\upsilon}, \upsilon)$. Equivalently, we can describe the transition function of a nondeterministic automaton as $\delta : Q \times \Sigma \times V \to 2^{((Q \times ACT)^{|\Upsilon|})}$. The tuple $\langle (q_1, act_1), \ldots, (q_{|\Upsilon|}, act_{|\Upsilon|})\rangle \in \delta(q, \sigma, A)$ is equivalent to the disjunct $(q_1, act_1, \upsilon_1) \wedge \ldots \wedge (q_{|\Upsilon|}, act_{|\Upsilon|}, \upsilon_{|\Upsilon|})$. When Υ is a singleton it is enough to consider the Büchi acceptance condition (as defined for NBW). We denote stack nondeterministic automata over infinite words by ST-NBW.

For an alternating stack automaton with 1-letter input alphabet, the location on the input tree and the structure of the input tree are not important. Hence, we can consider automata reading infinite words and we can write $\delta : Q \times V \to B^+(Q \times ACT)$. Accordingly, runs of such automata

$$\delta((c,q,s),A) = \begin{cases} \bigvee_{\substack{\langle s,A,act,s'\rangle \in T \\ \text{and } s' \in m}} \langle(\Diamond q,s'),act\rangle \vee \bigvee_{\substack{\langle s,A,act,s'\rangle \in T \\ \text{and } s' \in M}} \langle(q,s'),act\rangle & \text{If } c = \Diamond \\ \bigwedge_{\substack{\langle s,A,act,s'\rangle \in T \\ \text{and } s' \in m}} \langle(\Box q,s'),act\rangle \wedge \bigwedge_{\substack{\langle s,A,act,s'\rangle \in T \\ \text{and } s' \in M}} \langle\langle q,s'\rangle,act\rangle & \text{If } c = \Box \end{cases}$$

Figure 5.2: Transition of micro state

are $Q \times STORES$-labeled trees. We denote alternating stack parity automata with 1-letter input alphabet as ST-APW$_1$.

Harel and Raz show that the emptiness problem of nondeterministic Büchi stack automata on infinite trees is decidable in quintuply exponential time [HR94]. As we show in Section 5.5 their methods can be extended to ST-APW$_1$. We show that combining [HR94] and [KPV02a, KPV02b] gives an exponential algorithm for the emptiness of a ST-NBW and a double exponential algorithm for the emptiness of ST-APW$_1$.

Theorem 5.4.3
- *The emptiness of an ST-NBW N can be determined in time exponential in the size of N.*
- *The emptiness of an ST-APW$_1$ N can be determined in time double exponential in the size of N.*

5.4.2 Branching-Time Model Checking

We use the automata-theoretic approach to branching time model checking [KVW00]. Given an mMs system R and a graph automaton $\mathcal{A}$, we construct an ST-APW$_1$ N such that $\mathcal{L}(N) \neq \emptyset$ iff $G_R \models \mathcal{A}$.

The idea behind the construction is that the stack automaton N holds the control structure of $\mathcal{A}$ and the control structure of R within its finite control. When the control of R is in a micro state, our stack automaton mimics the behavior of R without changing the control state of $\mathcal{A}$. When the control of R is in a macro state, our stack automaton mimics a transition of $\mathcal{A}$ reading the new macro state. Formally we have the following.

Let $\mathcal{A} = \langle \Sigma, Q, \eta, q_0, \alpha\rangle$ be a graph automaton, $R = \langle \Sigma, V, S, L, T, s_0, \top, \bot\rangle$ be an mMs system where $S = m \cup M$, and $D = \{\varepsilon, \Diamond, \Box\}$. We construct the ST-APW$_1$ $N = \langle \{a\}, V, Q', \delta, q_0', \top, \bot, \alpha'\rangle$ where $Q' = (D \times Q \times S)$, the initial state $q_0' = (\varepsilon, q_0, s_0)$ and if $\alpha = \{F_1, ..., F_k\}$ (wlog k is odd) then $\alpha' = \{D \times F_1 \times M, \ldots, D \times F_k \times M, \{\Box\} \times Q \times m, \{\Diamond\} \times Q \times m\}$. We shorten notations by writing $(\Diamond q, s)$ instead of $(\Diamond, q, s)$ and similarly $(\Box q, s)$ and (q, s). The transition function δ is defined for every state $(c, q, s) \in D \times Q \times S$ and letter $A \in V$ as follows.

- For $c \in \{\Diamond, \Box\}$, the transition function is in Figure 5.2.
- For $c = \varepsilon$, then $s \in M$ is a macro state and we obtain $\delta((c, q, s), A)$ from $\eta(q, L(s, A))$ by replacing every atom $\Diamond q'$ in $\eta(q, L(s, A))$ by $\langle(\Diamond q', s), sp\rangle$, every atom $\Box q'$ by $\langle(\Box q', s), sp\rangle$, and every atom q' by $\langle(q', s), sp\rangle$.

$$\delta((q,s),A) = \begin{cases} \{\langle (q,s'), act\rangle \mid \langle s, A, act, s'\rangle \in T\} & s \in m \\ \left\{ \langle (q',s'), act\rangle \;\middle|\; \begin{array}{l} \langle s, A, act, s'\rangle \in T \\ \text{and } q' \in \eta(q, L(s,A)) \end{array} \right\} & s \in M \end{cases}$$

Figure 5.3: Transition of ST-NBW

Note that by including $\{\Box\} \times Q \times m$ as the maximal even set in α' and $\{\Diamond\} \times Q \times m$ as the maximal odd set in α' we ensure that when quantifying universally over successors, our ST-APW$_1$ does not care about infinite sequences of micro states and when quantifying existentially over successors, our ST-APW$_1$ does not allow to follow an infinite sequence of micro states. In Appendix 5.A we prove that $\mathcal{L}(N) \neq \emptyset$ iff $G_R \models \mathcal{A}$ by translating a run tree of N on the word a^ω to a run tree of $\mathcal{A}$ on G_R and vice versa.

Claim 5.4.4 *$\mathcal{L}(N) \neq \emptyset$ iff $G_R \models \mathcal{A}$.*

In Section 5.5 we show that the emptiness problem of an ST-APW$_1$ can be decided in double exponential time. Formally, we have the following.

Theorem 5.4.5 *The model checking problem of an mMs system and a graph automaton can be solved in time double exponential in both the size of the system and the size of the automaton.*

Combining Theorem 5.4.5 with Theorem 5.4.2 we get the following.

Corollary 5.4.6 *The model checking problem of a mu-calculus formula ψ and a micro-macro stack system R can be determined in time $2^{2^{O((|\psi||R|)^2)}}$.*

5.4.3 Linear-Time Model Checking

The μ-calculus is sufficiently strong to express all properties expressible in the linear temporal logic LTL (and in fact, all properties expressible by an ω-regular language) [Dam94]. In [KPV02a] we show that solving the emptiness of PD-NBW is simpler than solving the emptiness of a PD-ABT. In the next section we show that this is the case also for stack automata. Hence, we include a direct reduction from linear-time model checking to emptiness of ST-NBW.

We use the automata-theoretic approach to linear time model checking [VW86]. Given an mMs system R and an NBW N we wish to verify that $\mathcal{T}_{G_R} \cap \mathcal{L}(N) = \emptyset$. We construct a ST-NBW A such that $\mathcal{L}(A) \neq \emptyset$ iff $G_R \not\models A$ (i.e. $\mathcal{T}_{G_R} \cap \mathcal{L}(N) \neq \emptyset$).

The idea behind the construction is that the stack automaton A holds the control structure of N and the control structure of R within its finite control. When the control of R is in a micro state, our automaton mimics the behavior of R without changing the control state of N. When the control of R is in a macro state, our automaton mimics a transition of N reading the label of this macro state. Formally we have the following.

Let $N = \langle \Sigma, Q, q_0, \eta, F\rangle$ be an NBW and $R = \langle \Sigma, V, S, L, T, s_0, \top, \bot\rangle$ be an mMs system where $S = m \cup M$. We construct the ST-NBW $A = \langle \{a\}, V, Q \times S, \delta, (q_0, s_0), \top, \bot, F \times M\rangle$. The transition function δ is defined for every state $(q,s) \in Q \times S$ and letter $A \in V$ in Figure 5.3. Notice that a run can be accepting only if it visits infinitely often states in $F \times M$. Thus, a run of A corresponds

to a run of N on an infinite path in G_R. We show now that $\mathcal{L}(A) \neq \emptyset$ iff $G_R \not\models N$ by translating a run of A on the infinite word a^ω to a run of N on a trace in $\mathcal{T}_{G_R}$ and vice versa.

Claim 5.4.7 *$\mathcal{L}(A) \neq \emptyset$ iff $G_R \not\models N$.*

Proof: Suppose that $\mathcal{L}(A) \neq \emptyset$. Let $r = \langle (q_0, s_0), \top\uparrow\bot\rangle, \langle (q_1, s_1), z_1\rangle, \langle (q_2, s_2), z_2\rangle, \ldots$ be an accepting run of A on a^ω. Let $\langle (q_0, s_0), \top\uparrow\bot\rangle, \langle (q_{i_1}, s_{i_1}), z_{i_1}\rangle, \langle (q_{i_2}, s_{i_2}), z_{i_2}\rangle, \ldots$ be the subsequence of all states in r such that $s_j \in M$ is a macro state. It is easy to see that the sequence $\pi = (s_0, \top\uparrow\bot), (s_{i_1}, z_{i_1}), (s_{i_2}, z_{i_2}), \ldots$ is an infinite path in G_R and that the sequence $q_0, q_{i_1}, q_{i_2}, \ldots$ is an accepting run of N on the trace $L(s_0, \mathcal{H}(\top\uparrow\bot)) \cdot L(s_{i_1}, \mathcal{H}(z_{i_1})), \ldots \in \mathcal{T}_{G_R}$. Hence, $G_R \not\models N$.

In the other direction suppose that $G_R \not\models N$. Then there exists a path $\pi = (s_0, \top\uparrow\bot), (s_1, z_1), \ldots$ such that $r = q_0, q_1, \ldots$ is an accepting run of N on $L(\pi)$. As π is a path in G_R for every $i \geq 0$ there exists a sequence of micro states $(s^i_1, z^i_1), \ldots, (s^i_{n_i}, z^i_{n_i})$ such that $s^i_1 = s_i$, $z^i_1 = z_i$ and $s^i_{n_i} = s_{i+1}$ and $z^i_{n_i} = z_{i+1}$. We append the micro states between the micro states and construct the following run segments. For $i \geq 0$ let $r_i = \langle (q_{i+1}, s^i_2), z^i_2\rangle, \langle (q_{i+1}, s^i_3), z^i_3\rangle, \ldots, \langle (q_{i+1}, s^i_{n_i}), z^i_{n_i}\rangle$. As we start from (s^i_2, z^i_2) we know that $r = \langle (q_0, s_0), \top\uparrow\bot\rangle \cdot r_0 \cdot r_1 \cdots$ is a valid accepting run of A on a^ω. □

As our ST-NBW has 1-letter input alphabet, its emptiness is a special case of the emptiness of an ST-ABW$_1$. In Section 5.5 we show that the emptiness of ST-NBW can be solved in exponential time. Formally, we have the following.

Theorem 5.4.8 *The model checking of a mMs system and a nondeterministic Büchi automaton can be solved in time exponential in the size of the system and the automaton.*

Combining Theorem 5.4.8 with Theorem 5.4.1 we get the following.

Corollary 5.4.9 *Given an LTL formula φ and a micro-macro stack system R the model checking problem is solvable in time $2^{|R|^2 \cdot 2^{O(|\varphi|)}}$.*

5.5 Emptiness for Stack Automata

The emptiness problem of an automaton is to determine whether it accepts some input. In this section we solve the emptiness problem for ST-NBW and ST-APW$_1$ by a reduction to the emptiness of PD-NPW and PD-APW$_1$, respectively. Our reductions enhance the reduction of ST-NPT to PD-NPT given in [HR94]. An algorithm for checking the emptiness of PD-NBW and PD-APW$_1$ is given in [KPV02a, KPV02b].

Given a stack automaton we construct a pushdown automaton that records extra information on the pushdown store. This information is the summary of the stack automaton's actions when it reads the stack's contents. We want to know which parity sets are visited by the stack automaton and in what state does it 'get out' from the stack. In the case of ST-NBW, as the automaton is nondeterministic it may have many options for 'getting out'. Hence, for every state we record the possible states with which the stack automaton can surface after reading the contents of the stack. In the case of ST-APW$_1$, as the automaton is alternating, when reading the stack's contents it may spawn new copies of the automaton. Hence, for every state we record the possible sets of states with which the stack automaton can surface after reading the contents of the stack. We note that

the top-of-store symbol is a buffer between the possibility to change the contents of the stack and the meaningful information in the stack. Our pushdown automaton uses basically the same states as the stack automaton. However, it duplicates the set of states. One copy mimics the behavior of the stack automaton when reading the top-of-store symbol. The other copy mimics the behavior of the stack automaton when standing just before the top of the store and reading the last meaningful letter on the stack. We start with the simpler problem of solving the emptiness of ST-NBW and then extend it to the emptiness of ST-APW$_1$.

Notice that we can solve the emptiness of ST-NBW also in the case of alphabet that has more than one letter. For alternating pushdown automata (and hence also alternating stack automata) with alphabet with more than one letter emptiness is undecidable.

5.5.1 Emptiness for ST-NBW

We reduce the emptiness of an ST-NBW to the emptiness of PD-NBW. Consider an ST-NBW $S = \langle \Sigma, V, Q, \delta, q_0, \top, \bot, \alpha \rangle$ where $\alpha \subseteq Q$ and $\delta : Q \times \Sigma \times V \to 2^{Q \times ACT}$. It is easy to see that the emptiness problem of an ST-NBW can be reduced to that of an ST-NBW with one letter input alphabet. We assume that $|\Sigma| = 1$ and ignore this component. We construct a PD-NBW P such that $L(P) = \emptyset$ iff $L(S) = \emptyset$. The pushdown alphabet of P is $V \times \Gamma$ where $\Gamma = 2^{Q \times \{0,1\} \times (Q \cup \{acc\})}$. Thus, every letter in Γ is a $\{0,1\}$ labeled graph G on the set of vertices Q. For a state $q \in Q$ the set of nodes that are neighbors of q in G are the nodes reachable from q after a finite detour in the store. An infinite detour into the store is either accepting, in which case we add acc to the options of q or rejecting in which case we ignore it. Whenever state s chooses to enter the store in state q we choose one of the neighbors of q in G and move to it. Visits to the acceptance set are monitored using the label on the edges. Label 0 indicates that the path does not visit α and label 1 indicates that the path does visit α. More formally, we have the following.

Let $P = \langle \Sigma, V \times \Gamma, Q', \delta', q'_0, (\bot, \emptyset), \alpha' \rangle$ where the set of states is $Q' = (\{in, on\} \times Q \times \{0,1\}) \cup \{acc\}$, the initial state is $q'_0 = (in, q_0, 0)$, and the set of accepting states is $\alpha' = (\{in, on\} \times Q \times \{1\}) \cup \{acc\}$. The states marked by in serve as states that stand just before the top of the stack, the states marked by on serve as states that read the top-of-store symbol. In order to define the transition function δ' we define a function $f : V \times \Gamma \times V \to \Gamma$ that tells us how to extend the pushdown store. In case that on top of the pushdown store stands the pair (v, γ) and we want to emulate the extension of the stack with v' then we add to the pushdown store the pair $(v', f(v, \gamma, v'))$. Let $f(v, \gamma, v') = \gamma' \subseteq Q \times \{0,1\} \times (Q \cup \{acc\})$ such that $(t, i, t') \in \gamma'$ if there exists a sequence $r' = (j_1, s_1), (j_2, s_2), \ldots$ such that all the following hold.

- $(s_1, md) \in \delta(t, v')$ and $j_1 = 1$ iff $s_1 \in \alpha$.
- For every $1 \leq m < |r'|$ either (a) $(s_{m+1}, sp) \in \delta(s_m, v)$ and $j_{m+1} = 1$ iff $s_{m+1} \in \alpha$ or (b) $(s_m, j, s_{m+1}) \in \gamma$ and $j_{m+1} = j$.
- If r' is finite then either $(t', mu) \in \delta(s_{|r'|}, v)$ or $s_{|r'|} = acc$ and $t' = acc$.
- If r' is infinite then there are infinitely many m such that $j_m = 1$ and $t' = acc$.
- If there is some m such that $j_m = 1$ then $i = 1$. Otherwise $i = 0$.

The transition function $\delta' : Q' \times V \times \Gamma \to 2^{Q' \times \{sp, pop, push(z) \,:\, z \in V \times \Gamma\}}$ is defined as follows. For every $v \in V$ and $\gamma \in \Gamma$ we have $\delta'(acc, (v, \gamma)) = \{(acc, sp)\}$. For $q' = (on, q, i) \in Q'$, $v \in V$ and $\gamma \in \Gamma$ we have

- If $(q', sp) \in \delta(q, \top)$ then $((on, q', i'), sp) \in \delta'((on, q, i), (v, \gamma))$ and $i' = 1$ iff $q' \in \alpha$.
- If $(q', pop) \in \delta(q, \top)$ then $((on, q', i'), pop) \in \delta'((on, q, i), (v, \gamma))$ and $i' = 1$ iff $q' \in \alpha$.
- If $(q', push(v')) \in \delta(q, \top)$ then $((on, q', i'), push(v', f(v, \gamma, v'))) \in \delta'((on, q, i), (v, \gamma))$ and $i' = 1$ iff $q' \in \alpha$.
- If $(q', md) \in \delta(q, \top)$ then $((in, q', i'), sp) \in \delta((on, q, i), (v, \gamma))$ and $i' = 1$ iff $q' \in \alpha$.

For $(in, q, i) \in Q'$, $v \in V$, and $\gamma \in \Gamma$ we have

- If $(q', sp) \in \delta(q, v)$ then $((in, q', i'), sp) \in \delta'((in, q, i), (v, \gamma))$ and $i' = 1$ iff $q' \in \alpha$.
- If $(q', mu) \in \delta(q, v)$ then $((on, q', i'), sp) \in \delta'((in, q, i), (v, \gamma))$ and $i' = 1$ iff $q' \in \alpha$.
- If $(q, i', q') \in \gamma$ then
  - if $q' = acc$ then $(acc, sp) \in \delta'((in, q, i), (v, \gamma))$.
  - if $q' \neq acc$ then $((in, q', i'), sp) \in \delta'((in, q, i), (v, \gamma))$.

Prior to showing that S is empty iff P is empty we show that whenever the contents of the pushdown store is $w \in (V \times \Gamma)^*$ with the letter (v, γ) on top of the pushdown store (the first letter of w) and $(q, i, q') \in \gamma$ then we can find a partial run that connects the configuration (q, w') to configuration (q', w') where w' is the projection of w on its component in V (with $\top\uparrow$ concatenated on top). Formally, we have the following.

We extend the function f to a function $f : STORES \to (V \times \Gamma)^*$. For every $w = w_0, \ldots, w_m \in STORES$ we set $f(w)$ as follows.

- If $m = 3$ then $f(\top\uparrow\bot) = f(\uparrow\top\bot) = (\bot, \emptyset)$.
- If $m > 3$ where $w_i = \uparrow$ then $f(w) = (w_1, \gamma_1), \ldots, (w_{i-1}, \gamma_{i-1}), (w_{i+1}, \gamma_{i+1}), \ldots, (w_m, \gamma_m)$ where $\gamma_m = \emptyset$, $\gamma_{i-1} = f(w_{j+1}, \gamma_{j+1}, w_{j-1})$, and for every $j \neq i$ we have $\gamma_{j-1} = f(w_j, \gamma_j, w_{j-1})$. Note that the top-of-store symbol $\top$ and the head indicator $\uparrow$ are removed.

For every $w \in STORES$ where $w = w'\uparrow w''$ we say that the *head location* in w is $|w'|$, denoted $l(w)$. A sequence $(q_0, w_0), (q_1, w_1), \ldots$ is a *partial run* of S if for every i we have $(q_{i+1}, act) \in \delta(q_i, \mathcal{H}(w_i))$ and $w_{i+1} = \mathcal{B}(w_i, act)$. We say that pair $(i, q') \in \{0, 1\} \times (Q \cup \{acc\})$ is *reachable* from configuration (q, w) if there exists a partial run $r = (q_0, w_0), (q_1, w_1), \ldots$ such that $(q_0, w_0) = (q, w)$ and all the following holds.

- For all m we have $f(w_m) = f(w)$ (this means that w and w_m are the same but the location of $\uparrow$).
- If the run is finite then $q_{|r|} = q'$, and $i = 1$ iff there exists some m such that $q_m \in \alpha$.
- If the run is infinite then it visits α infinitely often, and $q' = acc$.

We say that (i, q') is *l-reachable* from (q, w) if in addition all the following hold.

- $l(w_0) = l$

- For every $1 \leq m < |r|$ we have $l(w_i) > l$.
- If the run is finite then $l(w_{|r|}) = l$.

Claim 5.5.1 *For every configuration (q, w) and pair $(i, q') \in \{0,1\} \times (Q \cup \{acc\})$ such that $f(w) = (w_1, \gamma_1), \ldots, (w_m, \gamma_m)$ we have (i, q') is $l(w)$-reachable from (q, w) iff $(q, i, q') \in \gamma_{l(w)}$.*

Proof: We prove the claim by induction on $m - l(w)$. If $m = l(w)$ then it must be the case that $(q', sp) \in \delta(q, \mathcal{H}(w))$. Suppose that $m > l(w)$. Let $r = (q_0, w_0), (q_1, w_1), \ldots$ denote the partial run. Let $r = (q_{j_0}, w_{j_0}), (q_{j_1}, w_{j_1}), \ldots$ denote the subsequence of locations where $l(w_{j_k}) = l(w) + 1$. For every k either $j_{k+1} = j_k + 1$ and $(q_{j_{k+1}}, sp) \in \delta(q_{j_k}, \mathcal{H}(w_{j_k}))$ or $j_{k+1} > j_k + 1$ and by induction $(q_{j_k}, i, q_{j_{k+1}}) \in \gamma_{l(w_{j_k})}$. We conclude that $(q, i, q') \in \gamma_{l(w)}$.

In the other direction we prove the claim by induction on $m - l(w)$. If $m = l(w)$ then we know that $\gamma_m = \emptyset$ and the claim follows. If $m > l(w)$ then we concatenate the partial runs promised by induction on $l(w) + 1$ to create a partial run connecting (q, w) and (i, q'). □

We are now ready to prove the following claim.

Claim 5.5.2 *$\mathcal{L}(S) = \emptyset$ iff $\mathcal{L}(P) = \emptyset$.*

Proof: Suppose that $\mathcal{L}(S) \neq \emptyset$. Then there exists an accepting run $r = (q_0, w_0), (q_1, w_1), \ldots$ on the word a^ω. We show that the subsequence of configurations where S stands on top of the store or just below the top of the store is an accepting run of P on a^ω. If the subsequence is finite then the run of P ends in an infinite chain of *acc*. More formally, we have the following.

Given a run $r = (q_0, w_0), (q_1, w_1), \ldots$ let $r' = (q_{i_0}, w_{i_0}), (q_{i_1}, w_{i_1}), \ldots$ denote the subsequence of configurations where S stands on top of the stack or one location before the end of the stack (i.e. $w_{i_j} = \uparrow \top w \bot$ or $w_{i_j} = \top \uparrow w \bot$ for some $w \in V^*$). We translate this subsequence into a run of P. Set $l_{i_j} = in$ if $w_{i_j} = \top \uparrow w \bot$ for some $w \in V^*$ and $l_{i_j} = on$ otherwise. Set $\alpha_{i_0} = 0$. Set $\alpha_{i_j} = 1$ if there exists k such that $i_{j-1} < k \leq i_j$ and $q_k \in \alpha$. Otherwise, set $\alpha_{i_j} = 0$. We claim that the sequence $r' = \langle (l_{i_0}, q_{i_0}, \alpha_{i_0}), f(w_{i_0}) \rangle, \langle (l_{i_1}, q_{i_1}, \alpha_{i_1}), f(w_{i_1}) \rangle, \ldots$ is a valid and accepting run of P on a^ω.

We first show that the transition $(\langle (l_{i_j}, q_{i_j}, \alpha_{i_j}), f(w_{i_j}) \rangle, \langle (l_{i_{j+1}}, q_{i_{j+1}}, \alpha_{i_{j+1}}), f(w_{i_{j+1}}) \rangle)$ is a legal transition of P. Clearly, if $i_{j+1} = i_j + 1$ then it is a legal transition. Suppose that $i_{j+1} > i_j + 1$. It follows from Claim 5.5.1 that it is a legal transition. We show that the run is accepting. If the run ends in an infinite sequence of *acc* then it is clearly accepting. Otherwise, α_{i_j} is set to 1 iff some state in $q_{i_j+1}, \ldots, q_{i_{j+1}}$ is accepting. As r is accepting so is r'.

In the other direction we extend the partial run of P to a full run of S using the promised partial runs from Claim 5.5.1. □

Theorem 5.5.3 [EHRS00, KPV02a] *The emptiness problem of a PD-NBW can be determined in time $O(|Q|^2 \cdot |\delta| \cdot |V|)$.*

Theorem 5.5.4 *The emptiness problem of a ST-NBW can be determined in time $|Q|^2 \cdot |\delta| \cdot |V| \cdot 2^{O(|Q|^2)}$.*

5.5.2 Emptiness for ST-APW$_1$

We advance now to the general case of ST-APW$_1$. First let us denote by *top-configurations* configurations in which the head indicator $\uparrow$ is either to the left or to the right of the top-of-store symbol $\top$ (i.e. configurations (q, w) where $w = \top\uparrow w'\bot$ or $w = \uparrow\top w'\bot$ for some $w' \in V^*$). Otherwise the configuration is denoted a *middle-configuration*. A nondeterministic word automaton has a single copy reading the input word at all times. Hence, when a nondeterministic automaton 'ventures' into the stack it comes out in a single copy (if at all). When a copy of the alternating automaton reads the last meaningful letter in the store, the run continues to form a finite tree whose internal nodes are all middle-configurations and whose leaves are top-configurations. On the pushdown store, we record for each state the possible sets of leaves of such trees (corresponding to *sp* and *mu* actions). The trees may be infinite if some path stays indefinitely inside the store. However we care only about the possible sets of leaves.

Consider an ST-APW$_1$ $S = \langle\{a\}, V, Q, \delta, q_0, \top, \bot, \alpha\rangle$ where $\alpha = \langle F_1, \ldots, F_k\rangle$ and $\delta : Q \times \{a\} \times V \to B^+(Q \times ACT)$. For a state $q \in Q$ let $r(q)$ denote the index i such that $q \in F_i$. We construct a PD-APW$_1$ P such that $\mathcal{L}(P) = \emptyset$ iff $\mathcal{L}(S) = \emptyset$. The pushdown alphabet of P is $V \times \Gamma$ where $\Gamma = 2^{Q \times 2^{([k] \times Q \times \{sp,mu\})}}$. Thus, every letter in Γ associates with every state in Q subsets of $[k] \times Q \times \{sp, mu\}$. For a state $q \in Q$ and $\gamma \in \Gamma$, the sets U such that $(q, U) \in \gamma$ are the possible transitions of state q. Again the automaton can enter the stack and stay there indefinitely. We just make sure that the set of states U participates in **some** run such that the paths that stay indefinitely in the stack are accepting. The minimal parity set visited is monitored via the labels attached to the states in Q. Label r indicates that the least rank visited along the detour is r. More formally, we have the following.

For a set $U \subseteq Q \times \{sp, mu\}$ let $r(U)$ denote the set $\{(r(s), s, \Delta) \mid (s, \Delta) \in U\}$. Let $\flat \subseteq Q \times 2^{[k] \times Q \times \{sp,mu\}}$ denote the set $\{(q, r(U)) \mid U \models \delta(q, \bot)\}$ of assignments that satisfy the transition of states in Q reading the bottom-of-store symbol $\bot$. The symbol $\flat$ is added to the store-bottom-symbol $\bot$ as the store-bottom-symbol of the pushdown automaton. Let $P = \langle\{a\}, V \times \Gamma, Q', \delta', q'_0, (\bot, \flat), \alpha'\rangle$ where the set of states is $Q' = \{in, on\} \times Q \times [k]$, the initial state is $q'_0 = (in, q_0, r(q_0))$, and the parity acceptance condition is $\alpha' = \langle\{in, on\} \times Q \times \{1\}, \ldots, \{in, on\} \times Q \times \{k\}\rangle$. The *in* and *on* states are just as in nondeterministic automata. The definition of the transition function and the extension of the pushdown store is quite technical. Details follow.

In order to define the transition function δ' we define a function $f : V \times \Gamma \times V \to \Gamma$ that tells us how to extend the pushdown store. In case that on top of the pushdown store stands the pair (v, γ) and we want to emulate the extension of the stack with v' then we add to the pushdown store the pair $(v', f(v, \gamma, v'))$. For every state $s \in Q$, in order to test whether to add the pair (s, U) to γ' where $U \subseteq [k] \times Q \times \{sp, mu\}$ we construct an APW that moves between the letters v and v'. The transition of states in U that read v' is set to **true** and states not in U that read v' are set to **false**. The pair (s, U) is added if the APW has an accepting run[4]. Formally, for every state $t \in Q$ and every subset $U \subseteq [k] \times Q \times \{sp, mu\}$ we define the APW $A^{t,U}_{v,\gamma,v'} = \langle\{a\}, Q'', \rho, t, \alpha''\rangle$ where the set of states is $Q'' = \{t\} \cup (\{v\} \times Q \times [k]) \cup (\{v'\} \times Q \times [k] \times \{sp, mu\})$, the parity acceptance condition is $\alpha'' = \langle\{v\} \times Q \times \{1\}, \ldots, \{v\} \times Q \times \{k\}\rangle$, and the transition function ρ is as follows[5].

- $\rho(t, a)$ is obtained from $\delta(t, v')$ by replacing (s, mu) by $(v', s, r(s), mu)$, replacing (s, sp) by

[4]This construction resembles the transformation of APW with ϵ-moves to APW without ϵ-moves in [Wil99].

[5]Note that the parity acceptance condition does not include states in $t \cup (\{v'\} \times Q \times [k] \times \{sp, mu\})$. These states cannot occur more than once on a path and adding / removing them from α'' does not matter.

$(v', s, r(s), sp)$, and replacing (s, md) by $(v, s, r(s))$.

- $\rho(v', s, r, \Delta) = \begin{cases} \textbf{true} & \text{if } (r, s, \Delta) \in U \\ \textbf{false} & \text{if } (r, s, \Delta) \notin U \end{cases}$
- $\rho(v, s, r) = \bigvee_{(s,U') \in \gamma} \bigwedge_{(r',s',\Delta) \in U'} c_r(r', s', \Delta)$ where $c_r : [k] \times Q \times \{sp, mu\} \to Q''$ is

  $c_r(r', s', \Delta) = \begin{cases} (v', min(r, r'), s', sp) & \text{if } \Delta = mu \\ (v, min(r, r'), s') & \text{if } \Delta = sp \end{cases}$

Finally, $f(v, \gamma, v') = \gamma'$ such that for every state $s \in Q$ and set $U \subseteq [k] \times Q \times \{sp, mu\}$ we have $(s, U) \in \gamma'$ iff $A^{s,U}_{v,\gamma,v'}$ has some accepting run [6].

The transition function $\delta' : Q' \times V \times \Gamma \to B^+(Q' \times \{sp, pop, push(z) \ : \ z \in V \times \Gamma\})$ is defined as follows. For every $v \in V$, and $\gamma \in \Gamma$ we have

- $\delta'((on, q, i), (v, \gamma))$ is obtained from $\delta(q, v)$ by replacing (s, sp) by $((on, s, r(s)), sp)$, replacing $(s, push(v'))$ by $((on, s, r(s)), push(v', f(v, \gamma, v')))$, replacing (s, pop) by $((on, s, r(s)), pop)$, and replacing (s, md) by $((in, s, r(s)), sp)$.
- $\delta'((in, q, i), (v, \gamma)) = \bigvee_{(q,U) \in \gamma} \bigwedge_{(r,s,\Delta) \in U} c(r, s, \Delta)$ where $c : [k] \times Q \times \{sp, mu\} \to Q'$ is as follows.

  $c(r, s, \Delta) = \begin{cases} (in, s, r) & \Delta = sp \\ (on, s, r) & \Delta = mu \end{cases}$

Prior to showing that S is empty iff P is empty we show that whenever the contents of the pushdown store is $w \in (V \times \Gamma)^*$ with the letter (v, γ) on top of the pushdown store (the first letter of w) and $(q, U) \in \gamma$ then we can find a run tree whose root is labeled by (q, w') and all its leaves are labeled by configurations $(s, \mathcal{B}(w', act))$ for $(r, s, act) \in U$ where w' is the projection of w on its component in V (with $\perp\uparrow$ concatenated on top) and all infinite paths in the tree are accepting according to α. Formally, we have the following.

We extend the function f to a function $f : STORES \to (V \times \Gamma)^*$ just like we did for ST-NBW. and we use the head location as defined for ST-NBW. A $Q \times STORES$-labeled tree $\langle T, r \rangle$ is a *partial run* of S if for every node $x \in T$ with $r(x) = (q, w)$ and $\delta(q, \mathcal{H}(w)) = \theta$ there exists a (possibly empty) set $\{(q_1, act_1), \ldots, (q_d, act_d)\} \models \theta$ and x has d successors $x_1, \ldots, x_d$ such that $r(x_j) = (q_j, \mathcal{B}(w, act_j))$. We say that a set $U \subseteq [k] \times Q \times \{sp, mu\}$ is *l-reachable* from configuration (q, w) if there exists a partial run $\langle T, r \rangle$ such that all the following hold.

- The root of T is labeled by $r(\epsilon) = (q, w)$.
- For every node $x \in T$ such that $r(x) = (s, w')$ then $f(w) = f(w')$ (this means that w and w' are the same but for the location of the head indicator $\uparrow$).
- For every node $x \in T$ such that $r(x) = (s, w')$ one of the following holds.
  - x is a leaf or $x = \epsilon$ and $l(w') = l(w)$ or $l(w') = l(w) - 1$.

[6] Notice that once a pair (s, U) is added to γ' then for every superset U' such that $U \subseteq U'$ we have $(s, U') \in \gamma'$. In practice, it is sufficient to add (s, U), however this would complicate the proof beyond necessary.

- x is internal and $x \neq \epsilon$ and $l(w') > l(w)$.

- For every leaf $x \in T$ such that $r(x) = (s, w')$ and the minimal rank on the path from the root to this leaf is r we have $(s, r, act) \in U$ and $w' = \mathcal{B}(w, act)$.

- Every infinite path in T is accepting according to α.

Claim 5.5.5 *For every configuration (q, w) and set $U \subseteq [k] \times Q \times \{sp, mu\}$ we have U is $l(w)$-reachable from (q, w) iff $(q, U) \in \gamma_{l(w)}$.*

Proof: Consider a configuration (q, w) and a set $U \subseteq [k] \times Q \times \{sp, mu\}$. Assume that U is $l(w)$-reachable from (q, w). There exists a partial run $\langle T, r\rangle$ such that all the leaves of T are labeled by states in U. We show by induction on $m - l(w)$ that $(q, U) \in \gamma_{l(w)}$. If $m = l(w)$ then the transition of (q, w) has to be supplied by states labeled by sp and mu. In this case, for the set U of q's successors we have $(q, U) \in \gamma_m = \flat$. Suppose $m > l(w)$. We define an equivalence relation on the nodes of T. Each equivalence class corresponds to the partial run that shows that some set U' is $l(w) + 1$-reachable from some configuration (q', w'). Then we use the induction assumption to show that $(q', U') \in \gamma_{l(w)+1}$ and prove that $(q, U) \in \gamma_{l(w)}$. Formally, we have the following.

Let $f(w) = (w_1, \gamma_1), \ldots, (w_m, \gamma_m)$. We abuse notation and for a node x such that $r(x) = (q', w')$ we write $l(x)$ for $l(w')$. Recall that every node x such that $l(x) \leq l(w)$ is either the root of T or a leaf. For every node x we add an annotation to x another node in T. If $l(x) \leq l(w)+1$ we annotate x by itself. If $l(x) > l(w) + 1$ then we annotate x by the least node x' such that $l(x') = l(w) + 1$ and there exists no $x' < x'' < x$ such that $l(x'') \leq l(w) + 1$. We say that two nodes x and x' are equivalent if the annotation of x and x' is equal.

For a node x such that $l(x) = l(w) + 1$ consider the tree T_x consisting of all the nodes in the equivalence class of x and their immediate descendants. That is, T_x includes internal nodes with head location greater than $l(w) + 1$ and leaves with head location at most $l(w) + 1$. Let $r(x) = (q', w')$ and let $U \subseteq [k] \times Q \times \{sp, mu\}$ be a set such that for every leaf $y \in T_x$ such that $r(y) = (q'', w'')$, r is the minimal rank on the path from the root to y, and $w'' = \mathcal{B}(w', act)$ then $(q'', r, act) \in U$. If y is a leaf in T_x then $l(y) = l(w) + 1$ or $l(y) = l(w)$. If follows that T_x is a partial run connecting (q', w') to U manifesting the fact that U is $l(w')$-reachable from (q', w'). As $l(w') = l(w) + 1$ we conclude that $(q', U) \in \gamma_{l(w)+1}$.

From above it is obvious that with every node x such that $l(x) = l(w) + 1$ we can associate a set $U_x \subseteq [k] \times Q \times \{sp, mu\}$ that 'labels' all the leaves of T_x. For every triplet (r, s, act) in U_x we choose one leaf $y^x_{r,s,act}$ in T_x that is 'labeled' by this triplet. Let T' be the minimal tree such that $\epsilon \in T'$ and for every node $x \in T'$ and every triplet $(r, s, act) \in U_x$ the leaf $y^x_{r,s,act} \in T'$. Let $U \subseteq [k] \times Q \times \{sp, mu\}$ denote the set of labels of leaves in T with the minimal ranks from the root to them. We claim that $\langle T', r'\rangle$ where r' is the restriction of r to T' is a run of $A^{q,U}_{w_{l(w)}, \gamma_{l(w)}, w_{l(w)+1}}$. From the explanation above it is clear that it is a valid run of $A^{q,U}_{w_{l(w)}, \gamma_{l(w)}, w_{l(w)+1}}$. We show that it is accepting. Every infinite path in T' visits infinitely many nodes x such that $l(x) = l(w) + 1$. However, an infinite path in T' corresponds to an infinite path in T. As the path in T is accepting we conclude that the path in T' is also accepting.

In the other direction, assume that $(q, U) \in \gamma_{l(w)}$. We prove by induction on $m - l(w)$ that U is $l(w)$-reachable from (q, w). For $m = l(w)$, we know that $\gamma_m = \flat$ and every pair $(t, U) \in \flat$ corresponds to a partial run that shows m-reachability of U from (q, w). Suppose $m > l(w)$. We use the induction assumption to replace every transition of $A^{q,U}_{w_{l(w)+1}, \gamma_{l(w)+1}, w_{l(w)}}$ by the partial run

that is promised from the membership of (q, U) in $\gamma_{l(w)+1}$. This is clearly a legal partial run that connects (q, w) to U. We have to show that the run is accepting. An infinite path that remains from some point onwards inside some partial run is definitely accepting. An infinite path that is the result of the concatenation of infinitely many partial runs is also accepting because every node is marked by the minimal rank between the root and the leaf. □

We are now ready to prove the following claim.

Claim 5.5.6 *$\mathcal{L}(S) = \emptyset$ iff $\mathcal{L}(P) = \emptyset$.*

Proof: From the previous claim it is clear that we can convert a run of S on a^ω to a legal run of P. Showing that this run is also accepting is not different from the arguments used in Claim 5.5.5.

The other direction is also similar. By popping the partial runs promised by Claim 5.5.5 we convert a run of P to a valid and accepting run of S. □

Theorem 5.5.7 [KPV02b] *The emptiness problem of a PD-APW$_1$ $P = \langle \{a\}, V, Q, \delta, q_0, \top, \bot, \alpha \rangle$ with n states and index k can be determined in time $(|V|nk)^{O((nk)^2)}$.*

Theorem 5.5.8 *The emptiness problem of a ST-APW$_1$ with n states and index k can be determined in time $2^{(nk)^2 \cdot 2^{O(n^2 k)}}$.*

5.6 Conclusions and Future Work

We propose a class of graphs called micro-macro stack graphs that strictly contains the class of prefix-recognizable graphs. We give direct automata-theoretic algorithms for model checking μ-calculus over micro-macro stack graphs. Our model checking algorithms is double exponential.

Since their introduction in [Cau96], prefix-recognizable graphs have been thoroughly studied. As a few examples we mention, games on prefix-recognizable graphs [Cac02], characterization of languages accepted by prefix-recognizable graphs [Sti00], and prefix-recognizable structures [Blu01]. There are many equivalent ways to represent prefix-recognizable graphs, using rewrite rules, as the outcome of regular restriction and inverse regular substitution on the infinite binary tree [Cau96], as monadic second order logic interpretations in the infinite binary tree [Blu01], and as graph equations [Cau96, Bar97]. All these issues need to be studied for mMs graphs.

As mentioned, the class of micro-macro stack graphs is contained in the class of high order pushdown graphs. As the monadic second-order theory of the latter is decidable [KNU03], it follows that the monadic second-order theory of micro-macro stack graphs is decidable.

5.7 Acknowledements

We thank T. Cachat for bringing [KNU03] to our attention and for clarifying the connection between mMs systems and high order pushdown systems.

Bibliography

[Bar97] K. Barthelmann. On equational simple graphs. Technical report, Universität Mainz, 1997.

[BCMS00] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. Unpublished manuscript, 2000.

[BE96] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1996.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, July 1997. Springer-Verlag.

[BLM01] P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessors using satisfiability solvers. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.

[Blu01] Achim Blumensath. Prefix-recognisable graphs and monadic second-order logic. Technical Report AIB-06-2001, RWTH Aachen, May 2001.

[BQ96] O. Burkart and Y.-M. Quemener. Model checking of infinite graphs defined by graph grammars. In *Proc. 1st International workshop on verification of infinite states systems*, volume 6 of *ENTCS*, page 15. Elsevier, 1996.

[BS95] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic J. Comut.*, 2:89–125, 1995.

[BS99] O. Burkart and B. Steffen. Model checking the full modal μ-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[Büc60] J.R. Büchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik und Grundl. Math.*, 6:66–92, 1960.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Bur97a] O. Burkart. Automatic verification of sequential infinite-state processes. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Lecture Notes in Computer Science*, volume 1354. Springer-Verlag, 1997.

[Bur97b] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. In F. Moller, editor, *Proc. 2nd International workshop on verification of infinite states systems*, 1997.

[Cac02] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. In *4th International Workshop on Verification of Infinite-State Systems*, *Electronic Notes in Theoretical Computer Science* 68(6), Brno, Czech Republic, August 2002.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 1996.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer-Verlag, 2001.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CM90] D. Caucal and R. Monfort. On the transition graphs of automata and grammars. In *16th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 484 of *Lecture Notes in Computer Science*, pages 311–337, Berlin, Germany, June 1990. Springer-Verlag.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[DW99] M. Dickhfer and T. Wilke. Timed alternating tree automata: the automata-theoretic solution to the TCTL model checking problem. In *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 281–290, Prague, Czech Republic, 1999. Springer-Verlag, Berlin.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247, Chicago, IL, July 2000. Springer-Verlag.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, San Juan, October 1991.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown automata. In F. Moller, editor, *Proc. 2nd International Workshop on Verification of Infinite States Systems*, 1997.

[GGH67a] S. Ginsburg, S.A. Greibach, and M.A. Harrison. One-way stack automata. *Journal of the ACM*, 14(2):381–418, 1967.

[GGH67b] S. Ginsburg, S.A. Greibach, and M.A. Harrison. Stack automata and compiling. *Journal of the ACM*, 14(1):172–201, 1967.

[HR94] D. Harel and D. Raz. Deciding emptiness for stack automata on infinite trees. *Information and Computation*, 113(2):278–299, September 1994.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *Proc. 20th International Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 552–562. Springer-Verlag, 1995.

[KNU03] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, Grenoble, France, April 2003. Springer-Verlag.

[KPV02a] O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer-Verlag, 2002.

[KPV02b] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *Proc. 9th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 2514 of *Lecture Notes in Computer Science*, pages 262–277. Springer-Verlag, 2002.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.

[KV00] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2000.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[Mor00] C. Morvan. On rational graphs. In *Proc. 3rd International Conference on Foundations of Software Science and Computation Structures*, volume 1784 of *Lecture Notes in Computer Science*, pages 252–266, Berlin, Germany, March 2000. Springer-Verlag.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Sti00] C. Stirgling. Decidability of bisimulation equivalence for pushdown processes. Unpublished manuscript, 2000.

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science*, pages 165–191, 1990.

[Tho01] W. Thomas. A short introduction to infinite automata. In *Proc. 5th. international conference on Developments in Language Theory*, volume 2295 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, July 2001.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th International Coll. on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer-Verlag, Berlin, July 1998.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer-Verlag, 1996.

[Wil99] T. Wilke. CTL^+ is exponentially more succinct than CTL. In C. Pandu Ragan, V. Raman, and R. Ramanujam, editors, *Proc. 19th conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1999.

5.A Proof of Claim 5.4.4

Proof: Suppose that N accepts the infinite word a^ω. Then there exists an accepting run tree $\langle T_r, r\rangle$ of N on a^ω. Recall that $\langle T_r, r\rangle$ is a $Q' \times STORES$ labeled tree. We prune the sequences of states of N whose labeling is in $(D \times Q \times m) \times STORES$, that is, states for which the component representing the state of R is in a micro state. We know that no such infinite sequence labeled by $\Diamond$ states exists (otherwise the run is not accepting). We construct an accepting run tree $\langle T'_r, r'\rangle$ of $\mathcal{A}$ on R as follows. To simplify the reading we denote nodes of the tree T_r by letters x and y, nodes of the tree T'_r by overline letters $\overline{x}$ and $\overline{y}$. The labeling $r(x)$ of a node in T_r is bounded by angles $\langle(\Diamond q, s), z\rangle$ and the labeling $r'(\overline{x})$ of a node in T'_r is bounded by square brackets $[(s, z), q]$. To every node $\overline{x}$ of T'_r we also attach an annotation, a node $x \in T_r$. We annotate the nodes so that if a node $\overline{x} \in T'_r$ is labeled by $r'(\overline{x}) = [(s, z), q]$ then the annotation x of $\overline{x}$ is labeled by $r(x) = \langle(q, s), z\rangle$. Recall that $\mathcal{A}$ reads only states of G_R, hence, s is a macro state.

We start from the root $\varepsilon \in T'_r$ and label it by $[(s_0, \top \uparrow \bot), q_0]$. We annotate the root by $\varepsilon \in T_r$. As $r(\varepsilon) = \langle(q_0, s_0), \top \uparrow \bot\rangle$ this conforms to our demands. Given a node $\overline{x} \in T'_r$ labeled by $r'(\overline{x}) = [(s, z), q]$ and annotated by x such that $r(x) = \langle(q, s), z\rangle$ we continue the tree T'_r below $\overline{x}$ as follows. Let $\theta = \delta((q, s), \mathcal{H}(z))$. Then there exists a set $Y \subseteq (D \times Q \times S) \times STORES$ such that $Y \models \theta$ and Y is the set used to continue the run tree $\langle T_r, r\rangle$ below the node x. The set Y' is obtained from Y by replacing an atom $\langle(c, q', s), sp\rangle$ by the atom (c, q'). Clearly, $Y' \models \eta(q, L(s, \mathcal{H}(z)))$. Every atom $\Diamond q' \in Y'$ corresponds to an atom $\langle(\Diamond q', s), sp\rangle \in Y$. In order to fulfill the atom $\langle(\Diamond q', s), sp\rangle$ the automaton N followed a sequence of micro states of R until reaching one macro state. We find this macro state and add it as a label of some successor of $\overline{x} \in T'_r$. Every atom $\Box q' \in Y'$ corresponds to an atom $\langle(\Box q', s), sp\rangle \in Y$. In order to fulfill the atom $\langle(\Box q', s), sp)$ the automaton N followed every possible sequence of micro states. When such a sequence of micro states reaches a macro state we add it as a label of some successor of $\overline{x} \in T'_r$. Finally, every atom $q' \in Y'$ corresponds to an atom $\langle(q', s), sp\rangle \in Y$. In order to fulfill the atom $\langle(q', s), sp\rangle$ the automaton N continued to a copy of the automaton in state (q', s) with the same store. We add one successor to $\overline{x}$ and label it accordingly. More formally we have the following.

For every atom $\langle(\Diamond q, s), sp\rangle \in Y$ there exists a successor x' of x such that $r(x') = \langle(\Diamond q, s\rangle, z\rangle$. According to the transition of $\Diamond$-states (states in $\{\Diamond\} \times Q \times S$), we know that every node y labeled by $\langle(\Diamond p, t), w\rangle$ has a successor labeled either by $\langle(\Diamond p, t'), w'\rangle$ where $t' \in m$ or by $\langle(p, t''), w''\rangle$ where $t'' \in M$. As there cannot exist infinite sequences of nodes labeled by $\Diamond$-states in the run of N we conclude that there must exists a descendant x'' of x' labeled by some state $\langle(q, t), w\rangle$ such that $t \in M$. Let x'' be the minimal such descendant. We add one successor $\overline{x}'$ to $\overline{x}$, label it by $[(t, w), q]$ and annotate it by x'. Clearly, the state (t, w) of G_R is a successor of the state (s, z) as the path in T_r from x to x'' corresponds to a sequence of micro states leading from (s, z) to (t, w).

For every atom $\langle(\Box q', s), sp\rangle \in Y$ there exists a successor x' of x such that $r(x') = \langle(\Box q', s), z\rangle$. According to the transition of $\Box$-states (states in $\{\Box\} \times Q \times S$), all successors of a node y labeled by $\langle(\Box p, t), w\rangle$ are labeled either by $\langle(\Box p, t'), w'\rangle$ where $t' \in m$ or by $\langle(p, t''), w''\rangle$ where $t'' \in M$. Hence, every path in the subtree under x' either is an infinite path labeled by $\{\Box q'\} \times S \times STORES$ or has a minimal node labeled by $\{q'\} \times S \times STORES$. Let Z denote the set of nodes $x'' > x'$ such that $r(x'') \in \{q'\} \times S \times STORES$ and there does not exist a node $x'' > y > x'$ such that $r(y) \in \{q'\} \times S \times STORES$. For each node $y' \in Z$ such that $r(y') = \langle(q', t), w\rangle$ we add a successor $\overline{y}$ to $\overline{x}$, label it by $[(t, w), q']$ and annotate it by y'. Clearly, the state (t, w) of G_R is a successor of the state (s, z) as the path in T_r from x to y' corresponds to a sequence of micro states leading from (s, z) to (t, w). Furthermore, for every successor (t', w') of (s, z) in G_R there exists a sequence

of micro states that is followed by N under the node $x' \in T_r$. Hence, there is some node in Z whose label is $\langle (q', t'), w' \rangle$ and a successor of $\overline{x} \in T'_r$ whose label is (t', w').

For every atom $\langle (q', s), sp \rangle \in Y$ there exists a successor x' of x such that $r(x') = \langle (q', s), z \rangle$, we add a successor $\overline{x}'$ of $\overline{x}$ such that $r'(\overline{x}') = [(s, z), q]$ and annotate $\overline{x}'$ by x'. Clearly, $[(s, z), q]$ fulfills the atom $q' \in Y'$.

We created a valid run tree of $\mathcal{A}$ on G_R. We have to show that it is accepting. Assume by contradiction that there exists an infinite rejecting path π in $\langle T'_r, r' \rangle$. Every node in π is annotated by some node in T_r. Thus, π induces an path $\pi' \subseteq T_r$ to which all the annotations of π belong. It is easy to see the path π' cannot end in an infinite sequence of nodes labeled by $(\{\Box, \Diamond\} \times Q \times m) \times STORES$. Furthermore, the only nodes in π' labeled by $(\{\varepsilon\} \times Q \times M) \times STORES$ are the nodes annotating nodes in π. As the acceptance condition α' of N was obtained from the acceptance condition α of $\mathcal{A}$ we conclude that π' must be rejecting as well.

In the other direction suppose that $\mathcal{A}$ accepts G_R. Then there exists an accepting run tree $\langle T'_r, r' \rangle$ of $\mathcal{A}$ on G_R. We construct an accepting run tree of N on a^ω by adding sequences of micro states between every two macro states in T'_r. We also add infinite sequences of micro states when necessary. The construction is similar to the above. $\Box$

Chapter 6

Liveness with Invisible Ranking

The method of Invisible Invariants was developed originally in order to verify safety properties of parameterized systems in a fully automatic manner. The method is based on (1) a *project&generalize* heuristic to generate auxiliary constructs for parameterized systems, and (2) *small model theorem* implying that it is sufficient to check the validity of logical assertions of certain syntactic form on small instantiations of a parameterized system. When proving safety properties of parameterized systems using the deductive proof rule INV, one can generate candidates for inductive assertions using *project&generalize*, and check their validity on small instantiations, concluding the validity of the safety property on any instantiation of the parameterized systems. The method earned the name *invisible invariants* since the candidate assertions are generated, embedded in the premises of INV, and checked for validity using BDD techniques without the user ever seeing the candidate invariants. The approach can be generalized to any deductive proof rule that (1) requires auxiliary constructs that can be generated by *project&generalize*, and (2) the premises resulting when using the constructs are of the form covered by the small model theorem.

This paper studies the problem of proving liveness properties of parameterized systems using the "invisible constructs" method. Starting with a proof rule and cases where the method can be applied almost "as is," the paper progresses to develop deductive proof rules for liveness and extend the small model theorem to cover many intricate families of parameterized systems.

6.1 Introduction

Uniform verification of parameterized systems is one of the most challenging problems in verification. Given a parameterized system $S(N) : P[1] \parallel \cdots \parallel P[N]$ and a property p, uniform verification attempts to verify that $S(N)$ satisfies p for every $N > 1$. One of the most powerful approaches to verification that is not restricted to finite-state systems is *deductive verification*. This approach is based on a set of proof rules in which the user has to establish the validity of a list of premises in order to validate a given temporal property of the system. The two tasks that the user has to perform are:

1. Provide some auxiliary constructs that appear in the premises of the rule;
2. Use the auxiliary constructs to establish the logical validity of the premises.

When performing manual deductive verification, the first task is usually the more difficult, requiring ingenuity, expertise, and a good understanding of the behavior of the program and the techniques

for formalizing these insights. The second task is often performed using theorem provers such as PVS [OSR93] or STeP [BBC+95], which require user guidance and interaction, and place additional burden on the user. The difficulties in the execution of these two tasks are the main reason why deductive verification is not used more widely.

A representative case is the verification of invariance properties using the proof rule INV of [MP95]: in order to prove that assertion r is an invariant of program P, the rule requires coming up with an auxiliary assertion φ that is *inductive* (i.e. is implied by the initial condition and is preserved under every computation step) and that strengthens (implies) r.

In [PRZ01, APR+01], we introduced the method of *invisible invariants*, that offers a method for automatic generation of the auxiliary assertion φ for parameterized systems, as well as an efficient algorithm for checking the validity of the premises of INV.

The generation of invisible auxiliary constructs is based on the following idea: it is often the case that an auxiliary assertion φ for a parameterized system $S(N)$ has the form $\forall i : [1..N].q(i)$ or, more generally, $\forall i \neq j.q(i,j)$. We construct an instance of the parameterized system taking a fixed value N_0 for the parameter N. For the finite-state instantiation $S(N_0)$, we compute, using BDDs, some assertion ψ that we wish to generalize to an assertion in the required form. Let r_1 be the projection of ψ on process $P[1]$, obtained by discarding references to variables that are local to all processes other than $P[1]$. We take $q(i)$ to be the generalization of r_1 obtained by replacing each reference to a local variable $P[1].x$ by a reference to $P[i].x$. The obtained $q(i)$ is our candidate for the body of the inductive assertion $\varphi : \forall i.q(i)$. We refer to this generalization procedure as *project&generalize*. For example, when computing invisible invariants, ψ is the set of reachable states of $S(N_0)$. The procedure can be easily generalized to generate assertions of the type $\forall i_1, \ldots, i_k.p(\vec{i})$.

Having obtained a candidate for the assertion φ, we still have to check the validity of the premises of the proof rule we wish to employ. Under the assumption that our assertional language is restricted to the predicates of equality and inequality between bounded-range integer variables (which is adequate for many of the parameterized systems we considered), we proved a *small-model* theorem, according to which, for a certain type of assertions, there exists a (small) bound N_0 such that such an assertion is valid for every N iff it is valid for all $N \leq N_0$. This enables using BDD-techniques to check the validity of such an assertion. The cases covered by the theorem are those whose premises can be written in the form $\forall\vec{i}\exists\vec{j}.\psi(\vec{i},\vec{j})$, where $\psi(\vec{i},\vec{j})$ is a quantifier-free assertion that may refer only to the global variables and the local variables of $P[i]$ and $P[j]$ ($\forall\exists$-*assertions* for short).

Being able to validate the premises on $S[N_0]$ has the additional important advantage that the user never sees the automatically generated auxiliary assertion φ. This assertion is produced as part of the procedure and is immediately consumed in order to validate the premises of the rule. Being generated by symbolic BDD-techniques, the representation of the auxiliary assertions is often extremely unreadable and non-intuitive, and it usually does not contribute to a better understanding of the program or its proof. Because the user never gets to see it, we refer to this method as the "method of *invisible invariants*."

As shown in [PRZ01, APR+01], embedding a $\forall\vec{i}.q(\vec{i})$ candidate inductive invariant in INV results in premises that fall under the small-model theorem. In this paper, we extend the method of invisible invariants to apply to proofs of the second most important class of properties – the class of *response properties*. Response properties are liveness properties that can be specified by the temporal formula $\Box(q \rightarrow \Diamond r)$ (also written as $q \Rightarrow \Diamond r$) and guarantees that every q-state is eventually followed by an r-state. To handle response properties, we consider a certain variant of

rule WELL [MP91], which establishes the validity of response properties under the assumption of *justice* (weak fairness). As is well known to users of this and similar rules, such a proof requires the generation of two kinds of auxiliary constructs: *helpful assertions* h_i, which characterize, for transition τ_i, the states from which the transition is helpful in promoting progress towards the goal (r), and *ranking functions*, which measure progress towards the goal.

In order to apply *project&generalize* to the automatic generation of the ranking functions, we propose a variant of rule WELL. In this variant rule, called DISTRANK, we associate, with each potentially helpful transition τ_i, an individual ranking function $\delta_i : \Sigma \mapsto [0..c]$, mapping states to integers in a small range $[0..c]$ for some fixed small constant c. The global ranking function can be obtained by forming the multi-set $\{\delta_i\}$. In most of the examples we consider, it suffices to take $c = 1$, which allows us to view each δ_i as an assertion, and generate it automatically using *project&generalize*.

If, when applying rule DISTRANK, the auxiliary constructs h_i and δ_i have no quantifiers, all the resulting premises are $\forall\exists$-premises and the small-model theorem can be used. One of the constructs required to be quantifier free are the helpful assertions, which characterize the set of states from which a given transition is helpful. Many simple protocols have helpful assertions that are quantifier-free (or, with the addition of some auxiliary variables, can be transformed into protocols that have quantifier-free helpful assertions). Some protocols, however, cannot be proven with such restricted assertions. To deal with such protocols, we extend the method of invisible ranking in two directions:

- Allowing expressions such as $i \pm 1$ to appear both in the transition relation as well as the auxiliary constructs; This is especially useful for ring algorithms, where many of the assertions have a $p(i, i+1)$ or $p(i, i-1)$ component.
- Allowing helpful assertions (and ranking functions) belonging to transitions of process i to be of the form $h(i) = \forall j.H(i, j)$, where $H(i, j)$ is a quantifier-free assertion; Such helpful assertions are common in "unstructured" systems where whether a transition of one process is helpful depends on the states of all its neighbors. Substituted in the standard proof rules for progress properties, these assertions lead to premises that do not conform to the required $\forall\exists$ form, and therefore cannot be validated using the small model theorem.

To handle the first extension we prove, in Subsection 6.6.1, a *modest model theorem*. The modest model theorem establishes that $\forall\exists$-premises containing $i \pm 1$ subexpressions can be validated on relatively small models. The size of the models, however, is larger when compared to the small model theorem of [PRZ01].

To handle the second extension, we introduce a novel proof rule, PRERANK: The main difficulty with helpful assertions of the form $h(i) = \forall j.H(i, j)$ is in the premise that claims that every "pending" state has some helpful transition enabled on it (D3 of rule DISTRANK in Section 6.2). Identifying such a helpful transition is the hardest step when applying the rule. The new rule, PRERANK (introduced in Section 6.7), implements a new mechanism for selecting a helpful transition based on the establishment of a *pre-order* among transitions in each state. The "helpful" transitions are identified as the transitions that are minimal according to this pre-order.

We emphasize that the two extensions are part of the same method, so that we can handle systems that both use ± 1 and require universal helpful assertions. For simplicity of exposition, we separate the extensions here.

Overview of Paper. In Section 6.2 we present the general computational model of FTS and the restrictions that enable the application of the invisible auxiliary constructs methods. We also review the small model theorem, which enables automatic validation of the premises of the various proof rules. In addition, we outline a procedure that replaces compassion requirements by justice requirements, which justifies our focus on proof rules that assume justice only. Section 6.3 introduces the new DISTRANK proof rule and explains how we automatically generate ranking and helpful assertions for the parameterized case. We refer to the new method as the method of *invisible ranking*. We use a version of the token ring protocol for an ongoing example in this section. Section 6.4 shows how to enhance the *project&generalize* method to enable the generation of invariants in the form of boolean combinations of universal assertions. This is demonstrated on a (different) version of the token ring protocol. In Section 6.5 we study a version of the Bakery algorithm, that seems beyond the scope of the invisible ranking method, and show how enhancing a protocol with some auxiliary variables can make it a suitable candidate for the method.

The method studied in Sections 6.3–6.5 is adequate for cases where the set of reachable states can be satisfactorily over-approximated by boolean combinations of $\forall$-assertions, and the helpful assertions as well as individual ranking functions δ_i can be represented by quantifier-free assertions. Not all examples can be handled by assertions which depend on a single parameter. In Section 6.6 we describe the *modest model theorem*, which allows handling of $i \pm 1$ expressions within assertions, and demonstrate these techniques on the Dining Philosopher problem. In Section 6.7 we present the PRERANK proof rule that uses pre-order among transitions, discuss how to automatically obtain the pre-order, and demonstrate the technique on the Bakery algorithm. Finally, we discuss the advantages of combining several pre-order relations, and demonstrate it on Szymanski's protocol for mutual exclusion [Szy88].

All our examples have been run on TLV [Sha00]. The interested reader may find the code, proof files, and output of all our examples in:

cs.nyu.edu/acsys/Tlv/assertions.

Related Work. This is the full version of [FPPZ04b, FPPZ04a]. See [ZP04] for a survey on the method of invisible constructs and an earlier version of invisible ranking.

The problem of uniform verification of parameterized systems is undecidable [AK86]. One approach to remedy this situation, pursued, e.g., in [EK00], is to look for restricted families of parameterized systems for which the problem becomes decidable. Unfortunately, the proposed restrictions are very severe and exclude many useful systems such as asynchronous systems where processes communicate by shared variables.

Another approach is to look for sound but incomplete methods. Representative works of this approach include methods based on: explicit induction [EN95], network invariants that can be viewed as implicit induction [LHR97], abstraction and approximation of network invariants [CGJ95], and other methods based on abstraction [GZ98]. Other methods include those relying on "regular model-checking" (e.g., [JN00]) that overcome some of the complexity issues by employing *acceleration* procedures, methods based on symmetry reduction (e.g., [GS97]), or compositional methods (e.g., ([McM98]), combining automatic abstraction with finite-instantiation due to symmetry. Some of these approaches (such as the "regular model checking" approach) are restricted to particular architectures and may, occasionally, fail to terminate. Others, require the user to provide auxiliary constructs and thus do not provide for fully automatic verification of parameterized systems.

Most of the mentioned methods only deal with safety properties. Among the methods dealing

with liveness properties, we mention [CS02], which handles termination of sequential programs, network invariants [LHR97], and *counter abstraction* [PXZ02].

6.2 Preliminaries

In this section we present our computational model, the small model theorem, and the procedure that allows to remove compassion (strong fairness).

6.2.1 Fair Transition Systems

As our computational model, we take a *fair transition system* (FTS) [MP95] $S = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, with:

- $V = \{u_1, \ldots, u_n\}$ — A finite set of typed *system variables*. A *state* s of the system provides a type-consistent interpretation of the system variables V, assigning to each variable $v \in V$ a value $s[v]$ in its domain. Let Σ denote the set of all states over V. An *assertion* over V is a first order formula over V. A state s satisfies an assertion φ, denoted $s \models \varphi$, if φ evaluates to T by assigning $s[v]$ to every variables v appearing in φ. We say that s is a φ-state if $s \models \varphi$.
- Θ — The *initial condition*: An assertion characterizing the initial states. A state is called *initial* if it is a Θ-state.
- $\mathcal{T}$ — A finite set of transitions. Every transition $\tau \in \mathcal{T}$ is an assertion $\tau(V, V')$ relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$. Given a state $s \in \Sigma$, we say that $s' \in \Sigma$ is a *τ-successor* of s if $\langle s, s' \rangle \models \tau(V, V')$ where, for each $v \in V$, we interpret v as $s[v]$ and v' as $s'[v]$. We say that transition τ is *enabled* in state s if it has some τ-successor, otherwise, we say that τ is *disabled* in s. Let $En(\tau)$ denote the assertion $\exists V'.\tau(V, V')$ characterizing the set of states in which τ is enabled, and let ρ denote the disjunction of all transitions, i.e. $\rho = \bigvee_{\tau \in \mathcal{T}} \tau$. The assertion ρ represents the *total transition* relation of S.
- $\mathcal{J} \subseteq \mathcal{T}$ — A set of *just* transitions (also called *weakly fair* transitions). Informally, $\tau \in \mathcal{J}$ rules out computations where τ is continuously enabled, but taken only finitely many times.
- $\mathcal{C} \subseteq \mathcal{T}$ — A set of *compassionate* transitions (also called *strongly fair* transitions). Informally, $\tau \in \mathcal{C}$ rules out computations where τ is enabled infinitely many times, but taken only finitely many times.

For technical reasons, and with no loss of generality, we assume that $\mathcal{T}$ always contains the *idling transition* $\tau_0 : V' = V$, which preserves the values of all system variables. Taking such a transition is often described as a *stuttering step*. We also require that the idling transition is taken to be a just transition.

Let $\sigma : s_0, s_1, s_2, \ldots$, be an infinite sequence of states. We say that transition $\tau \in \mathcal{T}$ is *enabled at position k* of σ if τ is enabled on s_k. We say that τ is *taken at position k* if s_{k+1} is a τ-successor of s_k. Note that several different transition can be considered as taken at the same position.

We say that σ is a *computation* of an FTS S if it satisfies the following requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.

- *Consecution* — For each $\ell = 0, 1, ...$, state $s_{\ell+1}$ is a ρ-successor of s_ℓ.
- *Justice* — for every $\tau \in \mathcal{J}$, it is not the case that τ is continuously enabled beyond some point j in σ (i.e., τ is enabled at every position $k \geq j$) but not taken beyond j.
- *Compassion* – for every $\tau \in \mathcal{C}$, it is not the case that τ is enabled at infinitely many positions in σ but taken at only finitely many positions.

Note that the idling transition being just implies that every computation contains infinitely many stuttering steps.

6.2.2 Bounded Fair Transition Systems

To allow the application of the invisible constructs methods, we further restrict the systems we study, leading to the model of *bounded fair transition systems* (BFTS), that is essentially the model of bounded discrete systems of [APR+01] augmented with fairness. For brevity, we describe here a simplified two-type model; the extension for the general multi-type case is straightforward.

Let $N \in \mathbf{N}^+$ be the *system's parameter*. We allow the following data types:

1. **bool**: the set of boolean and finite-range scalars;
2. **index**: a scalar data type that includes integers in the range $[1..N]$;
3. **data**: a scalar data type that includes integers in the range $[0..N]$; and
4. Any number of arrays of the type $\mathbf{index} \mapsto \mathbf{bool}$. We refer to these arrays as *boolean arrays*.
5. At most one array of the type $b : \mathbf{index} \mapsto \mathbf{data}$. We refer to this array as the *data array*.

Atomic formulas may compare two variables of the same type. E.g., if y and y' are **index** variables, and z is an $\mathbf{index} \mapsto \mathbf{data}$ array, then $y = y'$ and $z[y] < z[y']$ are both atomic formulas. For $z : \mathbf{index} \mapsto \mathbf{data}$ and $y : \mathbf{index}$, we also allow the special atomic formula $z[y] > 0$. We refer to quantifier-free formulas obtained by boolean combinations of such atomic formulas as *restricted assertions*.

As the initial condition Θ, we allow assertions of the form $\forall \vec{i}.u(\vec{i})$, where $u(\vec{i})$ is a restricted assertion.

As the transitions $\tau \in \mathcal{T}$, we allow assertions of the form $\tau(\vec{i}) : \forall \vec{j} : \psi(\vec{i}, \vec{j})$ for a restricted assertion $\psi(\vec{i}, \vec{j})$. This results in total transition $\rho : \exists \vec{i} : \forall \vec{j} : \psi(\vec{i}, \vec{j})$. For simplicity, we assume that all quantified and free variables are of type **index**.

Example 6.2.1 (The Token Ring Algorithm)
Consider program TOKEN-RING in Fig. 6.1, which is a mutual exclusion algorithm for any N processes.

In this version of the algorithm, the global variable *tloc* represents the current location of the token. Location 0 constitutes the non-critical section which may non-deterministically exit to the trying section at location 1. While being in the non-critical section, a process guarantees to move the token to its right neighbor, whenever it receives it. This is done by incrementing *tloc* by 1,

$$
\begin{array}{l}
\quad\quad\quad\quad \textbf{in} \quad N \;:\; \textbf{natural where } N > 1 \\
\quad\quad\quad\quad\quad\quad tloc \;:\; [1..N] \\
\quad\quad\quad\quad \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} 0: & \textbf{if } tloc = i \textbf{ then } tloc := i \oplus_N \mathbf{1} \\ & \textbf{go to } \{0,1\} \\ 1: & \textbf{await } tloc = i \\ 2: & \textbf{Critical} \end{array}\right] \end{array}\right] \\
\prod_{i=1}^{N} P[i] ::
\end{array}
$$

Figure 6.1: Program TOKEN-RING

$$
\begin{array}{lll}
V: & \left\{\begin{array}{ll} tloc: & [1..N] \\ \pi: & \textbf{array}[1..N] \textbf{ of } [0..2] \end{array}\right. & \\
\Theta: & \forall i.\pi[i] = 0 & \\
\mathcal{T}: & \left\{\begin{array}{lll} \tau_0^1(i): & \forall j \neq i: & \pi[i] = 0 \wedge tloc = i \wedge tloc' = i \oplus_N 1 \\ & & \wedge \pi'[i] \in \{0,1\} \wedge pres(\pi[j]) \\ \tau_0^2(i): & \forall j \neq i: & \pi[i] = 0 \wedge tloc \neq i \wedge \pi'[i] = 1 \wedge \\ & & pres(\pi[j], tloc) \\ \tau_1(i): & \forall j \neq i: & \pi[i] = 1 \wedge tloc = i \wedge \pi'[i] = 2 \wedge \\ & & pres(\pi[j], tloc) \\ \tau_2(i): & \forall j \neq i: & \pi[i] = 2 \wedge \pi'[i] = 0 \wedge pres(\pi[j], tloc) \\ \tau_{id}: & \forall j: & pres(\pi[j], tloc) \end{array}\right. & \\
\mathcal{J}: & \{\tau_0^1(i), \tau_1(i), \tau_2(i) \;\mid\; i \in [1..N]\} &
\end{array}
$$

Figure 6.2: BFTS for Program TOKEN-RING

modulo N. At the trying section, a process $P[i]$ waits until it receives the token which is signaled by the condition $tloc = i$.

Fig. 6.2 describes the BFTS corresponding to program TOKEN-RING, where for a variable $v \in V$, $pres(v)$ denotes $v' = v$ and for a set $U \subseteq V$, $pres(U)$ denotes $\bigwedge_{v \in U} pres(v)$. Note that $tloc$ is an **index**-variable, while the program counter π is an **index** $\mapsto$ **bool** array.

Strictly speaking, the transition relation as presented above does not conform to the definition of a boolean assertion since it contains the atomic formula $tloc' = i \oplus_N 1$. However, this can be rectified by a two-stage reduction. First, we replace $tloc' = i \oplus_N 1$ by $(i < N \;\wedge\; tloc' = i+1) \;\vee\; (i = N \;\wedge\; tloc' = 1)$. Then, we replace the formula $\tau(i) : \forall j \neq i : (\ldots tloc' = i+1 \ldots)$ by
$\tau(i, i_1) : \forall j \neq i, j_1 : (j_1 \leq i \;\vee\; i_1 \leq j_1) \;\wedge\; (\ldots tloc' = i_1 \ldots)$ which guarantees that $i_1 = i+1$.

Note that transition $\tau_0^2(i)$ is not listed as a just transition. This allows a process to remain forever in its non-critical location (0), as long as it diligently transfers any incoming token to its right neighbor. Also note that this system has an empty set of compassion transitions, which we omitted from the presentation in Fig. 6.2.

Example 6.2.2 (The Bakery Algorithm)
Consider program BAKERY *in Fig. 6.3, which is a variant of Lamport's original Bakery Algorithm that offers a solution to the mutual exclusion problem for any N processes.*

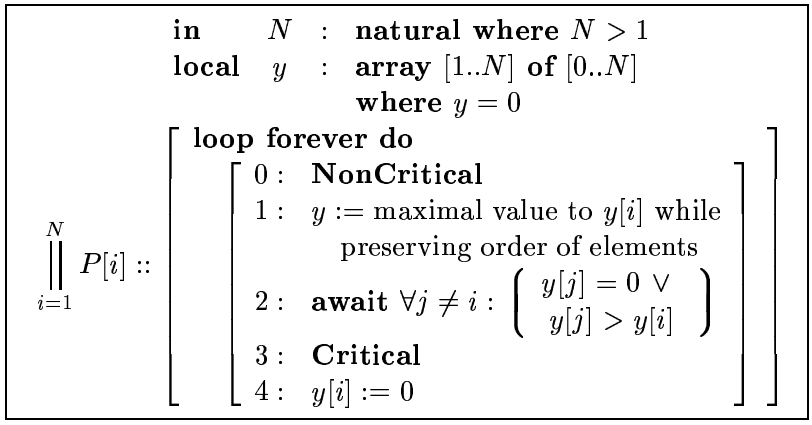


Figure 6.3: Program BAKERY

In this version of the algorithm, location 0 *constitutes the non-critical section which a process may nondeterministically exit to the trying section at location* 1. *Location* 1 *is the ticket assignment location. Location* 2 *is the waiting phase, where a process waits until it holds the minimal ticket. Location* 3 *is the critical section, and location* 4 *is the exit section. Note that* y, *the ticket array, is of type* **index** $\mapsto$ **data**, *and the program location array (which we denote by* π*) is of type* **index** $\mapsto$ **bool**. *Actually,* π *is of type* **index** $\mapsto [0..4]$, *but it can be encoded by three boolean arrays. Note also that the ticket assignment statement at* 1 *is non-deterministic and may modify the values of all tickets. Fig. 6.10 in Appendix 6.A.1 describes the* BFTS *corresponding to program* BAKERY.

Let α be an assertion over V, and R be an assertion over $V \cup V'$, which can be viewed as a transition relation. We denote by $\alpha \circ R$ the assertion characterizing all states which are R-successors of α-states. We denote by $\alpha \circ R^*$ the states reachable by an R-path of length zero or more from an α-state. In a symmetric way, we denote by $R \circ \alpha$ the assertion characterizing all the states which are R-predecessors of α-states.

6.2.3 The Small-Model Theorem

Let $\varphi : \forall\vec{i}\exists\vec{j}.R(\vec{i},\vec{j})$ be an AE-formula, where $R(\vec{i},\vec{j})$ is a restricted assertion which refers to the state variables of a parameterized BFTS $S(N)$ in addition to the quantified (**index**) variables $\vec{i}$ and $\vec{j}$. For simplicity, we assume that the only data variable/constant that may appear in R is the data constant 0. Let N_0 be the number of universally quantified variables, free **index** variables, and **index** constants appearing in R. The following theorem, stated first in [PRZ01] and extended in [APR$^+$01], provides the basis for the automatic validation of the premises in the proof rules.

Theorem 6.2.3 (Small model property)
Let φ *be an AE-formula as above. Then* φ *is valid over* $S(N)$ *for every* $N \geq 2$ *iff* φ *is valid over* $S(N)$ *for every* $N \leq N_0$.

For completeness of presentation we include the proof.

Proof: We denote by ψ the formula $\exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j})$, which is the negation of φ. Assume ψ is satisfiable in state s of a system $S(N_1)$ for $N_1 > N_0$. We show that it is satisfiable in a state s' of a system $S(N)$ for some $N \leq N_0$.

Let $\mathcal{V}_\exists$ be the set of **index** variables that appear existentially quantified in ψ. Let F be the set of **index** constants (including 1) and variables which appear free in ψ. Note that state s provides an interpretation for all the variables in F and all the arrays which appear in s. Similarly, let $\mathcal{V}_\forall$ be the set of **index** variables that appear universally quantified in ψ, i.e., the $\vec{j}$ variables.

The fact that $\psi : \exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j})$ is satisfiable in s means that there exists an assignment α which interprets all variables of $\mathcal{V}_\exists$ by values in the domain $[1..N_1]$ such that $(s,\alpha) \models \chi$, where $\chi : \forall\vec{j}.\neg R(\vec{i},\vec{j})$, and (s,α) is the joint interpretation which interprets all system variables according to state s and all $\mathcal{V}_\exists$-variables according to the assignment α.

Let $U = \{u_1 < u_2 < \cdots < u_k\}$ be a sorted list of values assigned to the $\mathcal{V}_\exists \cup F$-variables by α and s. Obviously, $k \leq N_0$. Let $f{:}\, U \to [1..k]$ be the bijection such that $f(u) = i$ iff $u = u_i$.

Similarly, let $D = \{0 = d_0 < d_1 < d_2 < \cdots < d_r\}$ be a sorted list of all the values assigned by s to the elements $b[u_i]$ for the data array b and $i \in [1..k]$. We always include 0 in D, even if it is not obtained as the value of some $b[u_i]$. Obviously, $r \leq k$. Let $g{:}\, D \to [1..r]$ be the bijection such that $g(d) = j$ iff $d = d_j$.

We construct a state s' of system $S(k)$ and an assignment $\alpha' : \mathcal{V}_\exists \mapsto [1..k]$, such that $(s',a') \models \chi$. The state s' is an interpretation defined as follows: For each variable $v \in F$, s' interprets v as $s'[v] = f(s[v])$. That is, $s[v] = u_i$ iff $s'[v] = i$. For every boolean array a : **index** $\mapsto$ **bool** we have $s'[a[i]] = s[a[u_i]]$, i.e., the value of $a[i]$ in state s' equals the value of $a[u_i]$ in state s. For the data array b : **index** $\mapsto$ **data**, we take $s'[b[i]] = g(s[b[u_i]])$, for each $i \in [1..k]$. That is, $s'[b[i]] = j$ iff $s[b[u_i]] = d_j$. Next, we define the interpretation α' as follows: For each variable $v \in \mathcal{V}_\exists$, α' interprets v as $\alpha'[v] = f(\alpha[v])$. That is, $\alpha[v] = u_i$ iff $\alpha'[v] = i$.

We proceed to show that $(s',\alpha') \models \chi$. To do so, consider an arbitrary assignment β' assigning to each variable $v \in \vec{j}$ a value $\beta'[v] \in [1..k]$. We will show that $(s',\alpha',\beta') \models \neg R(\vec{i},\vec{j})$. If this can be shown for every arbitrary assignment β', it follows that $(s',\alpha') \models \forall\vec{j}.\neg R(\vec{i},\vec{j})$. That is, $(s',\alpha') \models \chi$.

Consider the assignment β interpreting each $v \in \vec{j}$ as u_i iff $\beta'[v] = i$. It follows that β interprets each variable $v \in \vec{j}$ by a value in $[1..N_1]$. Since $(s,\alpha) \models \chi$, it follows that $(s,\alpha,\beta) \models \neg R(\vec{i},\vec{j})$. By induction on the structure of the formula $\neg R(\vec{i},\vec{j})$, we can show that every sub-formula $\gamma \in \neg R(\vec{i},\vec{j})$ evaluates to T under the joint interpretation (s,α,β) iff γ evaluates to T under the interpretation (s',α',β').

We conclude that $(s',\alpha') \models \chi$, which leads to the result that ψ is satisfied in the state s' of system $S(k)$. □

□

The small model theorem allows to check validity of AE-assertions on small models. In [PRZ01, APR+01] we obtain, using *project&generalize*, candidate inductive assertions for the set of reachable states that are A-formulae, checking their inductiveness required checking validity of AE-formulae, which can be accomplished, using BDD techniques.

6.2.4 Removing Compassion

The proof rule we are employing to prove progress properties assumes an incompassionate system (system with no compassionate transitions). As outlined in [KPP03][1] every FTS S can be converted

[1] The proof in [KPP03] is an adaptation of the proofs in [Cho74, Var91] to the case of transition systems.

into an incompassionate FTS $S_J = \langle V_J, \Theta_J, \mathcal{T}_J, \mathcal{J}_J, \emptyset \rangle$, where

$$\begin{array}{lcl} V_J & : & V \ \cup \ \{nvr_\tau : \ \textbf{boolean} \ \mid \ \tau \in \mathcal{C}\} \\ \Theta_J & : & \Theta \\ \mathcal{T}_J & : & \bigcup_{\tau \in \mathcal{T} \setminus \mathcal{C}} f_1(\tau) \ \cup \ \bigcup_{\tau \in \mathcal{C}} f_2(\tau) \\ \mathcal{J}_J & : & \bigcup_{\tau \in \mathcal{J} \setminus \mathcal{C}} f_1(\tau) \ \cup \ \bigcup_{\tau \in \mathcal{C}} f_2(\tau) \end{array}$$

where $f_1, f_2 : \mathcal{T} \to \mathcal{T}_J$ are defined by:

$$\begin{array}{lcl} f_1(\tau) & = & \tau \ \wedge \ pres(Nvr) \\ f_2(\tau) & = & \left(\begin{array}{l} \tau \wedge pres(Nvr) \ \vee \\ \neg nvr_\tau \wedge nvr'_\tau \wedge pres(V_J \setminus \{nvr_\tau\}) \end{array} \right) \\ Nvr & = & \{nvr_\tau \ \mid \ \tau \in \mathcal{C}\} \end{array}$$

This transformation adds to the system variables, for each compassionate transition τ, a new boolean variable nvr_τ. The intended role of nvr_τ is, non-deterministically, to identify a point in the computation beyond which τ is never enabled. The new transition relation includes two types of transitions: For each original non-compassionate transition τ, a transition $f_1(\tau)$ that behaves like τ while preserving the values of all nvr_τ variables. For each original compassionate transition $\tau \in \mathcal{C}$, $\mathcal{T}_J$ contains a transition $f_2(\tau)$ that either takes τ and preserves all nvr_τ variables, or changes nvr_τ from F to T and preserves all other variables. Intuitively, as long as $nvr_\tau = \text{F}$, $f_2(\tau)$ is enabled and, to comply with the justice requirement associated with $f_2(\tau)$, either τ is taken infinitely often, or nvr_τ eventually set to T. Once nvr_τ is set to T, τ is not expected to be enabled (and therefore taken) ever again.

Let Err denote the assertion $\bigvee_{\tau \in \mathcal{C}} (En(\tau) \ \wedge \ nvr_\tau)$, describing states where both τ is enabled and nvr_τ holds, which indicates that the prediction that τ will never be enabled is premature. For a computation σ_J of S_J, denote by $\sigma_J \Downarrow_V$ the sequence obtained from σ_J by projecting away the nvr variables. The relation between S and its compassion-free version S_J is stated by the following claim.

Claim 6.2.4 *Let σ be an infinite sequence of S-states. Then σ is an S-computation iff there exists an Err-free computation σ_J of S_J such that $\sigma_J \Downarrow_V = \sigma$.*

Proof: In one direction, let $\sigma = s_0, s_1, \ldots$ be a computation of S. We will show how to define the values of nvr_τ at each position of the computation, such the resulting sequence of S_J-states $\widetilde{\sigma} = \widetilde{s}_0, \widetilde{s}_1, \ldots$ is an Err-free computation of S_J.

The intention is to guarantee that transition $\tau \in \mathcal{C}$ is continuously disabled beyond some position j of σ iff nvr_τ is set to T at some position beyond j. For simplicity, assume that the compassionate transitions are $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$, and that we may refer to nvr_{τ_i} simply as nvr_i.

The initial values are determined as follows: for each $i = 1, \ldots, k$, the initial value of nvr_i is taken to be T iff τ_i is disabled at all positions of σ.

Next, we consider a step from position j to position $j+1$. If $s_j[V] \neq s_{j+1}[V]$ then we let $\widetilde{s}_{j+1}[Nvr] = \widetilde{s}_j[Nvr]$. That is, if at least one system variable of system S is modified in step j, then all the Nvr variables preserve their values.

On the other hand, if step j is a stuttering step, i.e. $s_j[V] = s_{j+1}[V]$, we search for a transition $\tau_i \in \mathcal{C}$ such that $\widetilde{s}_j[nvr_i] = \text{F}$ but τ_i is disabled at all positions beyond j. If there exists such a transition, let m be such a transition with the minimal index. We set $\widetilde{s}_{j+1}[nvr_m] = \text{T}$ and $\widetilde{s}_{j+1}[nvr_\ell] = \widetilde{s}_j[nvr_\ell]$, for all $\ell \neq m$.

If there does not exist a τ_i such as described above, we let again $\widetilde{s}_{j+1}[Nvr] = \widetilde{s}_j[Nvr]$.

Since, as previously observed, all computations contain infinitely many stuttering steps, the above definition guarantees that nvr_i eventually turns T iff τ_i eventually becomes continuously disabled. Furthermore, we never have a state in which τ_i is enabled while $nvr_i = \text{T}$.

In the other direction, consider a Err-free computation σ_j of S_j. We claim that $\sigma = \sigma_j \Downarrow_V$ is a computation of S. Suppose, by contradiction, that some $\tau \in \mathcal{C}$ is enabled infinitely often but taken only finitely often in σ. Then it must be the case that $f_2(\tau)$ is enabled infinitely often in σ_j. As τ is taken finitely often in σ it must be the case that nvr_τ is set in σ_j as not to violate $\mathcal{J}_j$. Since τ is enabled infinitely often, it is enabled after nvr_τ is increased and σ_j is not Err-free. □ □

We can therefore conclude that for every q and r,

$$S \models q \Rightarrow \diamondsuit r \quad \text{iff} \quad S_j \models (q \wedge \neg Err) \Rightarrow \diamondsuit(r \vee Err)$$

Which allows us to assume that all BFTSs we consider here have an empty compassion set.

6.3 The Method of Invisible Ranking

In this section we present a new proof rule that allows, in some cases, to obtain an automatic verification of liveness properties for an BFTS of any size. We first describe the new proof rule, and then present methods for the automatic generation of the auxiliary constructs required by the rule using TOKEN-RING as an ongoing example.

6.3.1 A Distributed Ranking Proof Rule

In Fig. 6.4 we present proof rule DISTRANK (DISTributed RANKing) for verifying response properties for BFTSs whose only fair transitions are just. The rule is configured to deal directly with parameterized systems. As in other rules for verifying response properties ([MP91], e.g.), progress is accomplished by the actions of *helpful transitions* in the system. In a parameterized system, the set of transitions has the structure $\mathcal{T}(N) = \{\tau_\ell[i] \mid \ell \in [0..m] \text{ and } i \in [1..N]\}$ for some fixed m. Typically, $[0..m]$ enumerates the locations within each process. For example, in program TOKEN-RING, $\mathcal{T}(N) = \{\tau_\ell[i] \mid \ell \in [0..2] \text{ and } i \in [1..N]\}$, where each transition $\tau_\ell[i]$ is associated with location $\ell \in [0..2]$ within process $i \in [1..N]$. Requiring that $\tau_\ell[i]$ is just guarantees that it is taken or disabled infinitely often, thus that $\tau_\ell[i]$ is not continuously enabled and never taken beyond some point.

Assertion φ is an invariant assertion characterizing all the reachable states. Assertion *pend* characterizes the states which can be reached from a reachable q-state by an r-free path. For each transition τ, assertion h_τ characterizes the states at which τ is *helpful*. These are the states s that have a τ-successor s', and the transition from s to s' leads to a progress towards the goal. This progress is observed by immediately reaching the goal or a decrease in the ranking function δ_τ, as stated in premises D5 and D6. The ranking functions δ_τ measure progress towards the goal. The

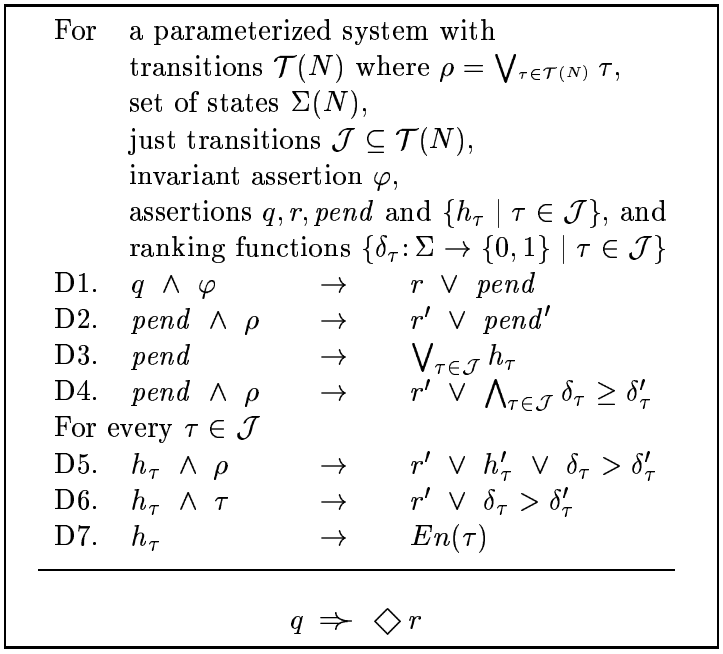


Figure 6.4: The liveness rule DISTRANK

disabling of τ is often caused by τ being taken (D6), but may also be caused by some condition turning false (D5). We require decrease in ranking in both cases.

Premise D1 guarantees that any reachable q-state satisfies r or *pend*. Premise D2 guarantees that any successor of a *pend*-state also satisfies r or *pend*. Premise D3 guarantees that any *pend*-state has at least one transition which is helpful in this state. Premise D4 guarantees that ranking never increases on transitions between two *pend*-states. Note that, due to D2, every ρ-successor of a *pend*-state that has not reached the goal is also a *pend*-state. Premise D5 guarantees that taking a step from an h_τ-state leads into a state which either already satisfies the goal r, or causes the rank δ_τ to decrease, or is again an h_τ-state. Premise D6 guarantees that taking a τ-transition from an h_τ-state either reaches the goal r or decreases the rank δ_τ. Premise D7 guarantees that in all h_τ-states τ is enabled. Together, premises D5, D6, and D7 imply that the computation cannot stay in h_τ forever, otherwise justice w.r.t τ is violated. Therefore, the computation must eventually decrease δ_τ. Since there are only finitely many δ_τ and until the goal is reached they monotonically decrease, we can conclude that eventually an r-state is reached.

6.3.2 Automatic Generation of the Auxiliary Constructs

We now proceed to show how the auxiliary constructs necessary for the application of rule DISTRANK can be automatically generated. Recall that we have to construct a *symbolic* version of each construct so that the rule can be applied to a generic N. We consider each auxiliary construct, provide a method for its generation, and illustrate it on the case of program TOKEN-RING.

In TOKEN-RING, the progress property we wish to check is:

$$\pi[z] = 1 \Longrightarrow \Diamond\, \pi[z] = 2$$

For simplicity, since all processes are symmetric we choose $z = 1$, thus, we check

$$\pi[1] = 1 \Longrightarrow \Diamond\, \pi[1] = 2$$

This property claims that every state in which process $P[1]$ is at location 1 is eventually followed by a state in which process $P[1]$ is at location 2.

The construction uses the instantiation $S(N_0)$ for the cutoff value N_0 required in Theorem 6.2.3. For TOKEN-RING, as explained in Subsection 6.3.3, $N_0 = 6$. We denote by Θ_C and ρ_C the initial condition and transition relation for $S(N_0)$. The construction begins by computing the *concrete* auxiliary constructs for $S(N_0)$, denoted by φ_C, $pend_C$. We then compute the concrete $h_k^C[j]$'s and $\delta_k^C[j]$'s. Next, we apply *project&generalize* to derive the symbolic (*abstract*) versions of these constructs: φ_A, $pend_A$, $h_k^A[j]$'s, and $\delta_k^A[j]$'s.

Since we focus on process 1, we would expect the constructs to have the symbolic forms $\varphi : \forall i.\varphi_A(i)$ and $pend : pend^A_{=1} \wedge \forall i{\neq}1.pend^A_{\neq 1}(i)$. For each $k \in [0..m]$, we need to compute $h_k^A[1], \delta_k^A[1]$, and the generic $h_k^A[i], \delta_k^A[i]$, that should be symbolic in i and apply for all i, $1 < i \leq N$. All generic constructs are allowed to refer to the global variables and to the variables local to $P[1]$ and $P[i]$.

Computing Concrete and Abstract φ: Compute on $S(N_0)$ the assertion $\varphi_C = reach_C = \Theta_C \circ \rho_C^*$ characterizing all states reachable within $S(N_0)$. Compute $\varphi_A(i) = reach_C[3 \mapsto i]$, by projecting $reach_C$ on index 3 , and then generalizing 3 to i. That is, maintaining only variables pertaining to process 3 and then replacing every reference to index 3 by a reference to index i.
For example, in TOKEN-RING(6),

$$\varphi_C = \bigwedge_{j=1}^{6} (at\_\ell_{0,1}[j] \vee tloc = j)$$

where $at\_\ell_{0,1}[j]$ is an abbreviation for $\pi[j] \in \{0, 1\}$.
The projection of φ_C on $j = 3$ yields

$$(at\_\ell_{0,1}[3] \vee tloc = 3)$$

The generalization of 3 to i yields

$$\varphi_A(i) : at\_\ell_{0,1}[i] \vee tloc = i$$

The assertion φ_A is $\forall i : \varphi_A(i)$.

Note that when we generalize, we should generalize not only the values of the variables local to $P[3]$ but also the case that the global variable, such as *tloc*, has the value 3. The choice of 3 as the generic value is arbitrary. Any other value would do as well, but we prefer indices different from $1, N$.

In this part we computed $\varphi_A(i)$ as the generalization of 3 into i in φ_C, which is denoted by $\varphi_A(i) = \varphi_C[3 \mapsto i]$. In later parts we may need to generalize two indices, such as $\alpha_A = \alpha_C[2 \mapsto i, 4 \mapsto j]$, where α_C and α_A are a concrete and abstract versions of some assertion α. The way we compute such abstractions over the state variables *tloc* and π of system TOKEN-RING is given by

$$\alpha_A(tloc, \pi) = i < j \ \wedge \ \exists tloc', \pi' : \begin{pmatrix} \alpha_C(tloc', \pi') \wedge \\ map(2, i, 4, j) \end{pmatrix}$$

where

$$map(2, i, 4, j) = \begin{pmatrix} \pi[i] = \pi'[2] \ \wedge \ \pi[j] = \pi'[4] & \wedge \\ tloc = i \iff tloc' = 2 & \wedge \\ tloc = j \iff tloc' = 4 & \wedge \\ tloc < i \iff tloc' < 2 & \wedge \\ tloc < j \iff tloc' < 4 & \end{pmatrix}$$

Note that this computation is very similar to the symbolic computation of the predecessor of an assertion, where $map(2, i, 4, j)$ serves as a transition relation. Indeed, we use the same module used by a symbolic model checker for carrying out this computation.

Computing Concrete and Abstract $pend$: Compute the assertion

$$pend_{_C} = (\varphi_{_C} \wedge q \wedge \neg r) \circ (\rho_{_C} \wedge \neg r')^*$$

characterizing all the states that can be reached from a reachable $(q \wedge \neg r)$-state by an r-free path. Then we take $pend^{A}_{=1} = pend_{_C}[1 \mapsto 1]$, and $pend^{A}_{\neq 1}(i) = pend_{_C}[1 \mapsto 1, 3 \mapsto i]$.

Thus, for TOKEN-RING(6),

$$pend_{_C} = \varphi_{_C} \ \wedge \ at\_\ell_1[1]$$

We therefore take

$$pend^{A}_{=1} : at\_\ell_1[1]$$

and

$$pend^{A}_{\neq 1}(i) : at\_\ell_1[1] \ \wedge \ (at\_\ell_{0,1}[i] \vee tloc = i)$$

Finally, $pend_{_A} = pend^{A}_{=1} \wedge \forall i{\neq}1 : pend^{A}_{\neq 1}(i)$, yielding

$$pend_{_A} = at\_\ell_1[1] \ \wedge \ \forall i{\neq}1 : (at\_\ell_{0,1}[i] \vee tloc = i).$$

Computing Concrete and Abstract $h_k[i]$'s: We compute the concrete helpful assertions $h^{C}_{k}[i]$. This is based on the following analysis: Assume that set is an assertion characterizing a set of states, and let τ be some just transition. We wish to identify the subset of states ϕ within set for which the transition τ is an *escape* transition. That is, any application of this transition to a ϕ-state takes us out of set. Consider the fix-point equation:

$$\phi \ = \ set \wedge En(\tau) \wedge AX(\phi \vee \neg set) \wedge AX_\tau(\neg set) \tag{6.1}$$

The equation states that every ϕ-state must satisfy $set \ \wedge \ En(\tau)$, every ρ-successor of a ϕ-state is either a ϕ-state or lies outside of set, and every τ-successor of a ϕ-state lies outside of set. Note that the expressions $AX\psi$ and $AX_\tau\psi$ can be computed by $\neg(\rho \circ (\neg\psi))$ and $\neg(\tau \circ (\neg\psi))$, respectively.

By taking the maximal solution of the fix-point equation (6.1), denoted $\nu\phi(set \wedge En(\tau) \ \wedge AX(\phi \vee \neg set) \ \wedge \ AX_\tau(\neg set))$, we compute the subset of states within set for which τ is helpful.

Following is an algorithm that computes the concrete helpful assertions $\{h^{C}_{k}[i]\}$ corresponding to the just transitions $\{\tau_k[i]\}$ of system $S(N_0)$. For simplicity, we will use $\tau \in \mathcal{T}(N_0)$ as a single parameter. Let

$$maxfix(set, \tau) \ : \ \nu\phi \left(\begin{array}{ll} set \wedge En(\tau) & \wedge \\ AX(\phi \vee \neg set) & \wedge \\ AX_\tau(\neg set) & \end{array} \right)$$

.

for each $\tau \in \mathcal{T}(N_0)$ **do** $h_\tau := 0$
$set := pend_{_C}$
for all $\tau \in \mathcal{T}(N_0)$ **s.t.** $maxfix(set, \tau) \neq 0$ **do**
$$\left[\begin{array}{l} h_\tau := h_\tau \ \vee \ maxfix(set, \tau) \\ set := set \ \wedge \ \neg h_\tau \end{array} \right]$$

The "**for all** $\tau \in \mathcal{T}(N_0)$" iteration terminates when it is no longer possible to find a $\tau \in \mathcal{T}(N_0)$ that satisfies the non-emptiness requirement. The iteration may choose the same τ more than once. When the iteration terminates, *set* is 0, i.e., for each of the states covered under $pend_C$ there exists a helpful justice requirement that causes it to progress.

Having found the concrete $h_k^C[i]$, we compute the abstract $h_k^A[i]$ by using *project&generalize* as follows: for each $k \in [0..m]$, we let $h_k^A[1] = h_k^C[1][1 \mapsto 1]$ and $h_k^A[i] = h_k^C[3][1 \mapsto 1, 3 \mapsto i]$.

Applying this procedure to TOKEN-RING(6), we obtain the symbolic helpful assertions described in Appendix 6.A.2.

Computing Concrete and Abstract $\delta_k[i]$'s: As before, we begin by computing the concrete ranking functions $\delta_k^C[i]$. We observe that $\delta_k^C[i]$ should equal 1 on every state for which $\tau_k[i]$ is helpful and should decrease from 1 to 0 on any transition that causes a helpful $\tau_k[i]$ to become unhelpful. Furthermore, $\delta_k^C[i]$ can never increase. It follows that $\delta_k^C[i]$ should equal 1 on every pending state from which there exists a pending path to a pending state satisfying $h_k^C[i]$. Thus, we compute $\delta_k^C[i] = pend_C \wedge ((\neg r)\, E\mathcal{U}\, h_k^C[i])$, where $E\mathcal{U}$ is the "existential-until" CTL operator. This formula identifies all states from which there exists an r-free path to an $(h_k^C[i])$-state.

Having found the concrete $\delta_k^C[i]$, we obtain the abstract $\delta_k^A[i]$ by using *project&generalize* as follows: for each $k \in [0..m]$, we let $\delta_k^A[1] = \delta_k^C[1][1 \mapsto 1]$ and $\delta_k^A[i] = \delta_k^C[3][1 \mapsto 1, 3 \mapsto i]$.

The abstract ranking function obtained by applying this procedure to TOKEN-RING(6)are described in Appendix 6.A.2.

6.3.3 Validating the Premises

Having computed internally the necessary auxiliary constructs, and checking the invariance of φ, it only remains to check that the six premises of rule DISTRANK are all valid for any value of N. Here we use the small model theorem stated in Theorem 6.2.3 which allows us to check their validity for all values of $N \leq N_0$ for the cutoff value of N_0 which is specified in the theorem. First, we have to ascertain that all premises have the required AE form. For auxiliary constructs of the form we have stipulated in this Section, this is straightforward. Next, we consider the value of N_0 required in each of the premises, and take the maximum. Note that once φ is known to be inductive, we can freely add it to the left-hand-side of each premise, which we do for the case of Premises D5, D6, and D7 that, unlike others, do not include any inductive component.

Usually, the most complicated premise is D2 and this is the one which determines the value of N_0. For program TOKEN-RING, this premise has the form (where we renamed the quantified variables to remove any naming conflicts):

$$\left\{ \begin{array}{c} (\forall a.pend(a)) \wedge \\ (\exists i, i_1 \forall j, j_1.\psi(i, i_1, j, j_1)) \end{array} \right\} \quad \rightarrow \quad r' \vee (\forall c.pend(c)),$$

which is logically equivalent to

$$\forall i, i_1, c\ \exists a, j, j_1. \left(\left(\begin{array}{c} pend(a) \wedge \\ \psi(i, i_1, j, j_1) \end{array} \right) \quad \rightarrow \quad r' \vee pend(c) \right)$$

The **index** variables which are universally quantified or appear free in the formula above, are $\{i, i_1, c, tloc, 1, N\}$ whose count is 6. It is therefore sufficient to take $N_0 = 6$. Having determined

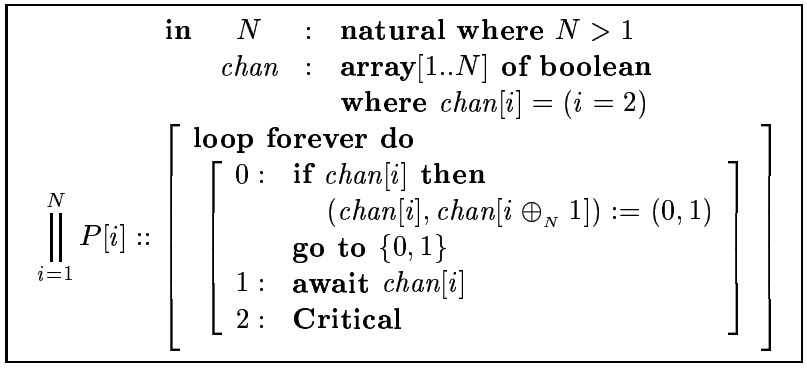


Figure 6.5: Program CHANNEL-RING

the size of N_0, it is straightforward to compute the premises of $S(N)$ for all $N \leq N_0$ and check that they are valid, using BDD symbolic methods.

We cannot use the same form of auxiliary constructs to automatically verify algorithm BAKERY(N), for every N. Indeed, it is straightforward to see that in order to conclude that $\tau_2[2]$ is helpful, one has to consider helpful assertions of the form $\forall j.\psi(i,j)$. In Section 6.7 we show how to obtain helpful assertions that relate to all processes and how to change the proof rule for such a case. We can still use the simple proof rule in order to automatically verify algorithm BAKERY(N). However, this requires the introduction of an auxiliary variable **minid** into the system, which is the index of the process which holds the ticket with minimal value. This is explained in detail in Section 6.5.

We emphasize that the generation of all assertions is *completely invisible*; so is the checking of the premises on the instantiated model. While the user may see the assertions, there is no need for the user to comprehend them. In fact, being generated using BDD techniques, they are often incomprehensible.

6.4 Cases Requiring an Existential Invariant

In some cases, A-assertions, i.e., assertions of the form $\forall i.u(i)$, are insufficient for capturing all the relevant features of the constructs φ_A and $pend_A$, and we need to consider assertions of the form $\forall i.u(i) \ \wedge \ \exists j.e(j)$. In this section we describe how to obtain constructs that are boolean combinations of A-assertions, illustrating the procedure and its applications on program CHANNEL-RING, presented in Fig. 6.5.

In this program the location of the token is identified by the index i such that $chan[i] = 1$. Computing the universal invariant according to the previous methods we obtain $\varphi_A : \forall i.(at\_\ell_{0,1} \vee chan[i])$, which is inductive but insufficient in order to establish the existence of a helpful transition for every pending state.

6.4.1 Generalizing *project&generalize*

We provide a sketch of the extension that enables computation of a $(\forall \wedge \exists)$ construct by obtaining a $\forall i.u(i) \ \wedge \ \exists j.e(j)$ invisible invariant. As before, we pick a value N_0, instantiate $S(N_0)$ and use the traditional *project&generalize* procedure to derive an inductive A-assertion $\varphi : \forall i.u(i)$. As a

byproduct of *project&generalize*, we compute $reach_C$ – the set of states reachable in $S(N_0)$. Being inductive and implied by the initial condition, the assertion φ is an over-approximation of $reach_C$. In order to isolate the (anticipated) assertion $e(j)$, we first compute the difference between the concrete reachable set and φ, denoted here by α_1. Obviously, we proceed only if α_1 is non-empty. Then, we *project&generalize* α_1 by replacing index 1 by k (α_2 below). Finally, we negate the result to get the proposed existential invariant (α_3 below).

Algorithm		
α_1	:=	$\bigwedge_{i=1}^{N_0} u(i) \ \wedge \ \neg reach_C$
α_2	:=	$\alpha_1[1 \mapsto k]$
α_3	:=	$\neg\alpha_2$

We use $\exists k.\alpha_3(k)$ as the candidate for an existential invariant. In the table below, we list the results of these computations for the case that $reach_C$ equals precisely the conjunction $\bigwedge_{i=1}^{N_0} w(i) \ \wedge \ \bigvee_{j=1}^{N_0} e(j)$ and the application of *project&generalize* to $reach_C$ yields precisely $u(i) = reach_C[1 \mapsto i] = w(i)$.

Results when $reach_C = \bigwedge_i w(i) \ \wedge \ \bigvee_j e(j)$		
α_1	=	$\bigwedge_i w(i) \ \wedge \ \bigwedge_j \neg e(j)$
α_2	=	$w(k) \ \wedge \ \neg e(k)$
α_3	=	$w(k) \rightarrow e(k)$

Note that, while we did not succeeded in precisely isolating $e(k)$, we computed instead the implication $w(k) \rightarrow e(k)$. However, the conjunction $\forall i.w(i) \ \wedge \ \exists k.(w(k) \rightarrow e(k))$ is logically equivalent to the conjunction $\forall i.w(i) \ \wedge \ \exists k.e(k)$.

This technique of obtaining an existential conjunct to an auxiliary assertion can be used for other auxiliary constructs.

6.4.2 Verifying Progress of CHANNEL-RING

Applying the generalized *project&generalize* to CHANNEL-RING we obtain, for the set of reachable states, the auxiliary construct:

$$\varphi_A : \left(\begin{array}{l} \forall i{\neq}k.(at\_\ell_{0,1} \vee chan[i]) \wedge \neg(chan[i] \wedge chan[k]) \\ \quad \wedge \\ \exists j.chan[j] \end{array} \right)$$

Using this extended form of an invariant for both φ_A and $pend_A$, we can complete the proof of program CHANNEL-RING using the methods of Section 6.3.

Applying the method of invisible ranking, with the new addition, to program CHANNEL-RING and the response property $at\_\ell_1[1] \Rightarrow \diamondsuit\, at\_\ell_2[1]$, we obtain, for example, $pend_A : at\_\ell_1[1] \ \wedge \ \varphi_A$, and for $i > 1$, $h_m^A[i] : at\_\ell_1[1] \ \wedge \ at\_\ell_m[i] \ \wedge \ chan[j]$. Thus, Premise D3 becomes:

$$\left(\begin{array}{l} at\_\ell_1[1] \\ \quad \wedge \\ \forall i{\neq}k.(at\_\ell_{0,1} \vee chan[i]) \wedge \neg(chan[i] \wedge chan[k]) \\ \quad \wedge \\ \exists j.chan[j] \end{array} \right) \rightarrow$$

$$at\_\ell_1[1] \ \wedge \ \exists j.chan[j]$$

which is obviously valid and has the AE form.

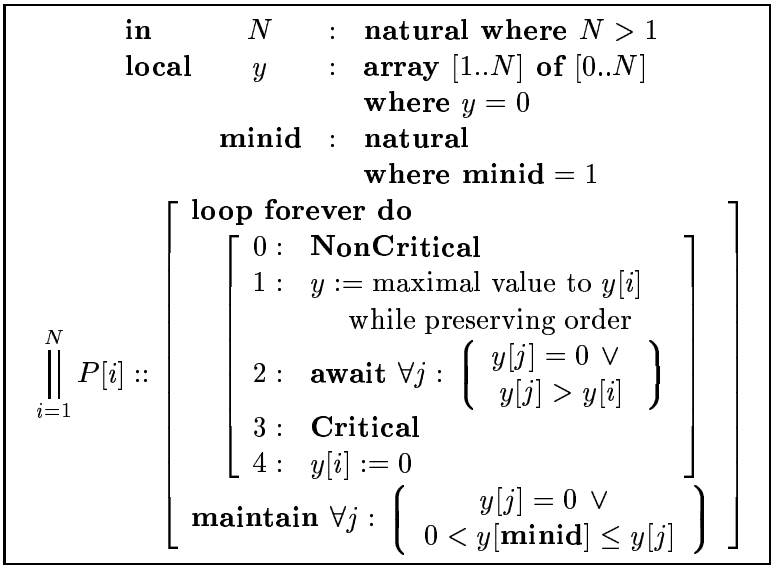


Figure 6.6: Program BAKERY with auxiliary variable **minid**

6.5 The Bakery Algorithm

As another example of the application of the invisible-ranking method we consider the modified version of program BAKERY, presented in Fig. 6.6.

As previously explained, in order to be able to use the rule in its current form we introduce the variable **minid**. The variable **minid** is expected to hold the index of a process whose y value is minimal among all the positive y-values. The **maintain** construct implies that this variable is updated, if necessary, whenever some y variables change their values. Already in [PRZ01] we pointed out that in some cases, it is necessary to add auxiliary variables in order to find inductive assertions with fewer indices. This version of BAKERY illustrates the case that such auxiliary variables may also be needed in the case of the invisible ranking method.

The property we wish to verify for this parameterized system is $at\_\ell_1[z] \Longrightarrow \diamondsuit\, at\_\ell_3[z]$ which implies accessibility for an arbitrary process $P[z]$.

Having the auxiliary variable **minid** as part of the system variables, we can proceed with the computation of the auxiliary constructs as explained in Section 6.3: After some simplifications, we can present the automatically derived constructs as detailed in Appendix 6.A.1. Using these derived auxiliary constructs, we can verify the validity of the premises of rule DISTRANK over $S(5)$ and conclude that for every value of N the property of accessibility holds.

6.6 Protocols with $p(i, i+1)$ Assertions

In algorithms for ring architectures, the auxiliary assertions for a process often depend, in addition to the process itself, on its immediate neighbors. Assume a ring of of size N. For every $j = 1, .., N$, denote $j \oplus 1 = (j \bmod N) + 1$ and $j \ominus 1 = ((j-2) \bmod N) + 1$. Assertions of the type $p(i, i \oplus 1)$ and $p(i, i \ominus 1)$ can be replaced by equivalent $\pm$-less AE-assertions. Unfortunately, this often results in formulae not covered by our small model theorem. We bypass the problem by establishing a new small model theorem that allows proving validity of $\forall\exists p(i, i \pm 1)$ assertions. The size of the model in

the new theorem is larger than the one indicated by the small model theorem, which is why we refer to it as "modest." We state the modest model theorem and prove it in Subsection 6.6.1, describe how to fine-tune the bounds in Subsection 6.6.2, and demonstrate its application in Subsection 6.6.3.

6.6.1 Modest Model Theorem

Consider a parameterized BFTS $S(N)$ with no **data** variables or arrays[2]. Let the formula φ: $\forall\vec{i}\exists\vec{j}.R(\vec{i},\vec{j})$ be an AE-formula, where $R(\vec{i},\vec{j})$ is a restricted assertion augmented by operators $\oplus 1$ and $\ominus 1$ which refer to quantified **index** variables $\vec{i}$ and $\vec{j}$. Let K be the number of universally quantified **index** variables, index constants (including 1 and N), and free **index** variables appearing in R. Assume there are ℓ **index** $\mapsto$ **bool** arrays in S and let $L = 2^\ell$, i.e., L is the number of different values that can be assigned to all variables indexed by a single process. Define $N_0 = (K-1)(L^2{+}1){+}K$.

Theorem 6.6.1 (Modest Model Theorem) *Let φ be an AE-formula as above. Then φ is valid over $S(N)$ for every $N \geq 2$ iff φ is valid over $S(N)$ for every $N \leq N_0$.*

Proof: We denote by ψ the formula $\exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j})$, which is the negation of φ. Assume ψ is satisfiable in state s of system $S(N_1)$ for $N_1 > N_0$. We show that ψ is also satisfiable in a state s' of a system $S(N)$ for some $N \leq N_0$.

Let $\mathcal{V}_\exists$ be the set of **index** variables that appear existentially quantified in ψ. Let F be the set of **index** constants (including 1 and N) and variables which appear free in ψ. Note that state s provides an interpretation for all the variables in F. Observe that $|\mathcal{V}_\exists \cup F| = K$. Similarly, let $\mathcal{V}_\forall$ be the set of **index** variables that appear universally quantified in ψ, i.e., the $\vec{j}$ variables.

The fact that $\psi : \exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j})$ is satisfiable in s means that there exists an assignment α which interprets all variables of $\mathcal{V}_\exists$ by values in the domain $[1..N_1]$ such that $(s,\alpha) \models \chi$, where $\chi : \forall\vec{j}.\neg R(\vec{i},\vec{j})$, and (s,α) is the joint interpretation which interprets all system variables according to state s and all $\mathcal{V}_\exists$-variables according to the assignment α.

Let $U = \{1 = u_1 < u_2 < \cdots < u_k = N_1\}$ be a sorted list of values assigned to the $F \cup \mathcal{V}_\exists$-variables by the joint interpretation (s,α).

Since $N_1 > N_0$ there exist some $i < k$ such that $u_{i+1}{-}u_i > L^2{+}1$. We construct a state s', in an instantiation $S(N')$, $N' < N_1$, such that $s' \models \psi$. The process is repeated until we obtain an instantiation that satisfies ψ where the u's are at most $L^2{+}1$ apart from one another.

Since $u_{i+1}{-}u_i > L^2{+}1$, there exist two pairs of adjacent indices between u_i and u_{i+1} that agree on their local array values, i.e., there exist some m and n such that $u_i{<}m{<}n < n+1 < u_{i+1}$ and, for every boolean array a: **index** $\mapsto$ **bool**, we have $a[m] = a[n]$ and $a[m{+}1] = a[n{+}1]$. Intuitively, removing all processes $m{+}1,\ldots,n$ does not impact any of the other processes whose indices are in U, since the array values of their immediate neighbors remain the same. In particular, since $m{+}1$ and $n{+}1$ are identical, processes m and $n{+}1$ maintain the same neighbors after the removal. Once the processes are removed, the remaining processes are renumbered.

Formally, let $N' = N_1{-}(n{-}m)$, and define the function g: $[1..N_1] \to [1..N']$ such that $g(i) = i$ for $i \leq m$, and $g(i) = i{-}(n{-}m)$ for $i \geq n{+}1$. It is easy to see that g is injective and onto, hence g^{-1} is well defined. Consider the state s' of system $S(N')$ such that for every array a : **index** $\mapsto$ **bool** we have $s'[a[i]] = s[a[g^{-1}(i)]]$, i.e., the value of a in state s' at index i is the value of a in state s at index $g^{-1}(i)$.

[2]This assumption is here for simplicity's sake and can be removed at the cost of increasing the bound.

We proceed to show that $(s', \alpha') \models \chi$. To do so, consider an arbitrary assignment β' assigning to each variable $v \in \vec{j}$ a value $\beta'[v] \in [1..N']$. We will show that $(s', \alpha', \beta') \models \neg R(\vec{i}, \vec{j})$. If this can be shown for every arbitrary assignment β', it follows that $(s', \alpha') \models \forall \vec{j}. \neg R(\vec{i}, \vec{j})$. That is, $(s', \alpha') \models \chi$.

Consider the assignment β interpreting each $v \in \vec{j}$ as r, $r \in [1..N_1]$ iff $\beta'[v] = g(r)$. Since $(s, \alpha) \models \chi$, it follows that $(s, \alpha, \beta) \models \neg R(\vec{i}, \vec{j})$. By induction on the structure of the formula $\neg R(\vec{i}, \vec{j})$, we can show that every sub-formula $\gamma \in \neg R(\vec{i}, \vec{j})$ evaluates to T under the joint interpretation (s, α, β) iff γ evaluates to T under the interpretation (s', α', β').

We conclude that $(s', \alpha') \models \chi$, which leads to the result that ψ is satisfied in the state s' of system $S(N')$.

Thus, s' is obtained from s by leaving the values of the **index** variables in the range $1..m$ intact, reducing the **index** variables larger than n by $n - m$, while maintaining the assignments of their **index** $\mapsto$ **bool** variables. Obviously, s' is a state of $S(N_1-(n-m))$ that satisfies ψ. □ □

6.6.2 Calibrating N_0

The bound computed in Theorem 6.6.1 may be quite large. In some cases it can be reduced significantly as we explain below.

General bool's: If there are **index** $\mapsto$ **bool** arrays for arbitrary (finite) **bool**, L in the bound should be replaced by the product of the sizes of ranges of all **index** $\mapsto$ **bool** variables.

Primed Occurrences: When a variable appears both unprimed and primed in $R(.)$, both occurrences add to the count (unless equal). This is in general the case with the transition relation ρ (that appears on the l-h-s of several implicants in our proof rules). While it may seem that each additional variable that can be modified doubles the count, only a single step is to be considered at a time, which is further restricted by *reach* (*reach* appears explicitly in all the implicants; moreover, it can always be added since it is shown to be an invariant). Hence, in practice, the bound can often be reduced as to be manageable.

Restricted Use of $\pm$: Assume that for each $\mathcal{V}_\forall$ variable under a $\pm$ operator, all occurrences of the operator are of the same kind (only $\oplus$ or $\ominus$ for each variable). Then, when reducing a large model into a smaller one, instead of finding two processes at the endpoint of a chain that agree on values of both their neighbors, it suffices to find a pair that agrees on one neighbor, which implies a chain of length L. Consequently, in this case the cut-off value is $N_0 = (K-1)L+K$. Further analysis reveals that if only one operator ($\oplus$ or $\ominus$) is applied to $\mathcal{V}_\exists$ variables, then the bound can be further reduced to $N_0 = (K-1)(L-1)+K$.

Restricting to "Observable" States: Suppose that a process only has a "partial" view of its neighbor, i.e., can access some, but not all, of its neighbor **index** $\mapsto$ **bool** array entries. Then, it suffices to find processes that agree on the part of the state observable by their neighbors, and not the complete state.

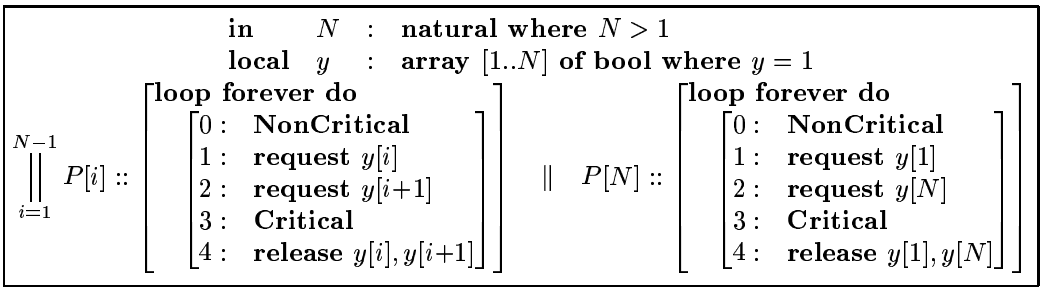


Figure 6.7: Program DINE: Solution to the Dining Philosophers Problem

Chains of Consecutive Free Variables: If, in addition to $N, 1$ there are longer, or other, chains of consecutive values the bound is reduced accordingly, since there are less "gaps" to collapse. E.g., when there is a $N-1, N, 1$ combination, the $(K-1)$ in the bound can be replaced by $(K-2)$.

6.6.3 Example: Dining Philosophers

We demonstrate the use of the modest model theorem by validating accessibility for a classical solution to the dining philosophers problem, using rule DISTRANK.

Consider program DINE that offers a solution to the dining philosophers problem for any N philosophers. The program uses semaphores for forks. In this program, $N-1$ philosophers (processes $P[1], \ldots, P[N-1]$) reach first for their left forks and then for their right forks, while $P[N]$ reaches first for its right fork and only then for its left fork. Program DINE is presented in Fig. 6.7.

The semaphore instructions "**request** x" and "**release** x" appearing in the program stand, respectively, for "$\langle$**when** $x = 1$ **do** $x := 0\rangle$" and "$x := 1$". Consequently, the transition associated with "**request** x", is compassionate, indicating that if a process is requesting a semaphore that is available infinitely often, it obtains it infinitely many times.

As outlined in Section 6.2.4, we transform the BFTS into a compassion-free BFTS by adding two new boolean arrays, nvr_1 and nvr_2, each $nvr_\ell[i]$ corresponding to the request of process i at location ℓ. Appendix 6.A.3 describes the BFTS we associate with Program DINE.

The progress property of the original system is

$$(\pi[z] = 1) \Rightarrow \diamondsuit(\pi[z] = 3)$$

which is proved in two steps, the first establishing that $(\pi[z] = 1) \Rightarrow \diamondsuit(\pi[z] = 2)$ and the second establishing that $(\pi[z] = 2) \Rightarrow \diamondsuit(\pi[z] = 3)$. For simplicity of presentation, we restrict discussion to the latter progress property.

Since $P[N]$ differs from $P[1], \ldots, P[N-1]$, and since it accesses $y[1]$, which is also accessed by $P[1]$, and $y[N]$, which is also accessed by $P[N-1]$, we choose some z in the range $2, \ldots, N-2$ and prove progress of $P[z]$. The progress property of the other three processes can be established separately (and similarly.) Taking into account the translation into a compassion-free system, the property we attempt to prove is

$$(\pi[z] = 2) \Rightarrow \diamondsuit(\pi[z] = 3 \ \vee \ Err) \qquad (2 \leq z \leq N-2)$$

where

$$Err = \left(\begin{array}{ll} \bigvee_{i=1}^{N-1}(\pi[i] = 1 \ \wedge \ y[i] \ \wedge \ nvr_1[i]) & \vee \\ \bigvee_{i=2}^{N}(\pi[i-1] = 2 \ \wedge \ y[i] \ \wedge \ nvr_2[i-1]) & \vee \\ (\pi[N] = 1 \ \wedge \ y[1] \ \wedge \ nvr_1[N]) & \vee \\ (\pi[N] = 2 \ \wedge \ y[N] \ \wedge \ nvr_2[N]) & \end{array} \right)$$

6.6.4 Automatic Generation of Symbolic Assertions

Following the guidelines in Section 6.3, we instantiate the program DINE according to the small model theorem, compute the auxiliary *concrete* constructs for the instantiation, and abstract them. Here, we chose an instantiation of $N_0 = 6$ (obviously, we need $N_0 \geq 4$; it seems safer to allow at least a chain of three that does not depend on the "special" three, hence we obtained 6.) For the progress property, we chose $z = 3$, and attempt to prove $(\pi[3] = 2) \Rightarrow\!\!\!\!\!\!\rightarrow \Diamond(\pi[3] = 3)$. Due to the structure of Program DINE, process $P[i]$ depends only on its neighbors, thus, we expect the auxiliary constructs to include only assertions that refer to two neighboring process at the same time. We chose to focus on pairs of the form $(i, i \ominus 1)$.

We first compute $\varphi_{A}(i, i \ominus 1)$, which is the abstraction of the set of reachable states. We distinguish between three cases, $i = 1$, $i = N$, and $i = 2, \ldots, N-1$. For the first, we take $\varphi^{A}_{=1} = reach_{C}[1 \mapsto 1, 6 \mapsto N]$ (i.e., project the concrete $reach_{C}$ on 1 and 6 and generalize to 1 and N), for the second, we take $\varphi^{A}_{=N} = reach_{C}[6 \mapsto N, 5 \mapsto N-1]$ (i.e., project the concrete $reach_{C}$ on 6 and 5 and generalize to N and $N-1$), and for the third we take $\varphi^{A}_{\neq 1,N} = reach_{C}[3 \mapsto i, 2 \mapsto i-1]$ (i.e., project the concrete $reach_{C}$ on 3 and 2 and generalize to i and $i-1$). The abstract pending sets we obtain are in Appendix 6.A.3. We then define:

$$\varphi_{A} \ = \ \varphi^{A}_{=1} \ \wedge \ \varphi^{A}_{=N} \ \wedge \ \forall i \notin \{1, N\} : \varphi^{A}_{\neq 1,N}(i, i-1)$$

and define $pend_{A} \ = \ \varphi_{A} \ \wedge \ \neg Err \ \wedge \ \pi[3] = 2$.

For the helpful sets, and the δ's, we obtain, as expected, assertions of the type $p(i, i \ominus 1)$. The assertions are described in Appendix 6.A.3.

Thus, the proof of inductiveness of φ, as well as all premises of DISTRANK are now of the form covered by the modest model theorem.

We now compute the size of the instantiation needed. Premises D1, D3, and D7 relate only to unprimed copies of the variables. Other premises relate to both unprimed and primed copies of the variables. When we use the modest model theorem "as is" the resulting figures are $L = 40^2 = 1600$ (5 possible locations, one fork, two *nvr* variables, all counted as current and next), $L^2{+}1 \sim 2.5 \cdot 10^6$ which results in a bound of about 10^7 processes. In order to get a reasonable figure we use the following reductions.

- We syntactically analyze all the resulting assertions and find that only variables in $\mathcal{V}_{\exists}$ are referenced by both $\oplus 1$ and $\ominus 1$. Variables in $\mathcal{V}_{\forall}$ are referenced only by $\ominus 1$. Thus, we have to search only for two identical processes and not for two pairs of adjacent processes.
- The transition ρ is on the left-hand-side of the implication in all the premises that include primed variables (D2,D4,D5, and D6). This implies that all possible counter-examples to these premises satisfy ρ. According to ρ all primed variables for every $j \notin \{i, i \oplus 1\}$ equal to their unprimed versions. Thus, if we treat $i, i \oplus 1$ as another 2-element long chain of universally quantified variables, we do not have to consider different values of the primed variables. It follows that we can use $L = 40$ for our search for duplicate entries.

Construct	BDD nodes
φ	1,779
$pend$	3,024
ρ	10,778
h_ℓ's	< 10
δ_ℓ's	≤ 10

Premise	Time to Validate
φ (inductiveness)	0.39 seconds
D1	< 0.01 seconds
D2	0.42 seconds
D3	0.01 seconds
D4	163.74 seconds
D5+D6	138.59 seconds
D7	0.02 seconds

Table 6.1: Run time and space results for DINE

As a result, the value L above (the maximal length of chain with no "equivalent" processes) is 40. There are three free variables in the system, 1, N, and $N-1$. (The reason we include $N-1$ is, e.g., its explicit mention in φ_A). Following the remarks on the modest model theorem, since the three variables are consecutive, and since with all universally quantified variables we use only $i \ominus 1$, the size of the (modest) model we need to take is $40(u+1)+u+4$, where u is the number of universally quantified variables. Since $u \leq 2$ for each of D1–D7 (it is 0 for D4, 1 for D1, and 2 for D2, D3, and D5), it is sufficient to choose an instantiation of 128.[3]

In Table 6.1, we present the number of BDD nodes computed for each auxiliary construct, and the time it took to validate each of the inductiveness of φ and all the premises (D1–D7) on the largest instantiation (128 philosophers). Checking all instantiations (2-128) took less than 8 hours.

6.7 Imposing Ordering on Transitions

Sections 3–4 dealt with helpful transitions $h_k[i]$ (and ranking functions) which depended only on the single index i. In the previous section we showed how to extend this approach to the case in which $h_k[i]$ may also depend on indices $i \ominus 1$ and $i \oplus 1$. In this section we study helpful assertions that depend on all $j \neq i$. Such multiple-index helpful assertions appear quite frequently. As a matter of fact most helpful assertions seem to be of the type $h(i) : \forall j.p(i,j)$ where i is the index of the process which can take a helpful step, and all other processes (j) satisfy some supporting conditions. However, such a helpful assertion presents a problem when trying to verify premise D4 of rule DISTRANK, since we obtain an EA-disjunct in the premise. In this section we show a new proof rule for progress, that allows us to order the helpful assertions in terms of the precedence of their helpfulness. "The helpful" assertion is then the minimal in the ordering, so that we can avoid the disjunction in the r-h-s of Premise D4.

[3]By modifying *project&generalize* to include only part of the variables of a process and not all variables this can be further reduced to 83 processes.

For a parameterized system with a transition $\mathcal{T} = \mathcal{T}(N)$
set of states $\Sigma(N)$, just transitions $\mathcal{J} \subseteq \mathcal{T}(N)$,
invariant assertion φ,
assertions $q, r, pend$ and $\{h_\tau \mid \tau \in \mathcal{J}\}$,
ranking functions $\{\delta_\tau : \Sigma \to \{0,1\} \mid \tau \in \mathcal{J}\}$,
and a pre-order $\preceq : \Sigma \mapsto 2^{\mathcal{T}\times\mathcal{T}}$

R1.	$q \ \wedge \ \varphi$	$\rightarrow$	$r \ \vee \ pend$
R2.	$pend \ \wedge \ \rho$	$\rightarrow$	$r' \ \vee \ pend'$
R3.	$pend \ \wedge \ \rho$	$\rightarrow$	$r' \ \vee \ \bigwedge_{\tau\in\mathcal{J}} \delta_\tau \geq \delta'_\tau$

For every $\tau \in \mathcal{J}$

R4.	$pend \ \wedge \ \left(\bigwedge_{\tau_1 \in \mathcal{J}} \tau \preceq \tau_1 \right)$	$\rightarrow$	h_τ
R5.	$h_\tau \ \wedge \ \rho$	$\rightarrow$	$r' \ \vee \ h'_\tau \ \vee \ \delta_\tau > \delta'_\tau$
R6.	$h_\tau \ \wedge \ \tau$	$\rightarrow$	$r' \ \vee \ \delta_\tau > \delta'_\tau$
R7.	h_τ	$\rightarrow$	$En(\tau)$
R8.	$pend$	$\rightarrow$	$\tau \preceq \tau$

For every $\tau_1, \tau_2 \in \mathcal{J}$

R9.	$pend \ \wedge \ \tau \preceq \tau_1 \ \wedge \ \tau_1 \preceq \tau_2$	$\rightarrow$	$\tau \preceq \tau_2$
R10.	$pend$	$\rightarrow$	$\tau \preceq \tau_1 \ \vee \ \tau_1 \preceq \tau$

$$q \Rightarrow \diamondsuit r$$

Figure 6.8: The liveness rule PreRank

6.7.1 Pre-Ordering Transitions

A binary relation $\preceq$ is a pre-order over domain $\mathcal{D}$ if it is reflexive, transitive, and total.

Consider an BFTS S with set of transitions $\mathcal{T}(N) = [0..m] \times N$ (as in Subsection 6.3.1). For every state in $S(N)$, define a pre-order $\preceq$ over $\mathcal{T}$. From the totality of $\preceq$, every $S(N)$-state has some $\tau_\ell[i] \in \mathcal{T}$ which is minimal according to $\preceq$. We replace premise D4 in DistRank with a premise stating that that for every pending state s, the transition that is minimal in s is also helpful at s. We call the new rule PreRank and, to avoid confusion, name its premises R1–R7. Thus, PreRank is exactly like DistRank, with the addition of a pre-order $\preceq : \Sigma \to 2^{\mathcal{T}\times\mathcal{T}}$, premises ascertaining that the relation $\preceq$ is a pre-order (R8–R10), and replacement of D4 by R4. In order to automate the application of PreRank, we need to be able to automatically generate the pre-order relation $\preceq$. As usual, we first instantiate $S(N_0)$, compute concrete $\preceq_C$, and then use the method *project&generalize* to compute an abstract $\preceq_A$. The main problem is the computation of the concrete $\preceq_C$. We define $s \models \tau_1 \preceq \tau_2$ if $s \models \Phi(\tau_1, \tau_2)$ for the following CTL formula:

$$\Phi(\tau_1, \tau_2) : \left\{ \begin{array}{l} \mathbf{A}((\neg h_{\tau_2} \wedge pend) \, \mathcal{W} \, h_{\tau_1}) \ \vee \\ \neg\mathbf{A}((\neg h_{\tau_1} \wedge pend) \, \mathcal{W} \, h_{\tau_2}) \end{array} \right\} \tag{6.2}$$

where $\mathcal{W}$ is the *weak-until* or *unless* operator.

The intuition behind the first disjunct is that for a state s, transition τ_1 is "helpful earlier" than τ_2 if every path departing from s doesn't reach h_{τ_2} before it reaches h_{τ_1}. The role of the second disjunct is to guarantee the totality of $\preceq$, so that when τ_1 becomes helpful earlier than τ_2 in some computations, and τ_2 precedes τ_1 in others, we obtain both $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_1$. To abstract a formula $\mathbf{A}(\varphi(h^C_k[i]) \, \mathcal{W} \, \psi(h^C_m[j]))$, we compute the assertion $\mathbf{A}(\varphi(h^C_k[2]) \, \mathcal{W} \, \psi(h^C_m[3]))$ over $S(N_0)$ (2

and 3 being chosen arbitrarily to represent two generic indices), and then generalize 2 to i and 3 to j. To abstract the negation of such a formula, we first abstract the formula, and then negate the result. Therefore, to abstract Formula (6.2), we abstract each $\mathbf{A}\,\mathcal{W}$-formula separately, and then take the disjunction of the first abstract assertion with the negation of the second abstract assertion.

6.7.2 Case Study: Bakery

Consider program BAKERY of Example 6.2.2 (Fig. 6.3). Suppose we want to verify $(\pi[z] = 1) \Rightarrow \Diamond(\pi[z] = 3)$. We instantiate the system to $N_0 = 3$, and obtain the auxiliary assertions φ, *pend*, the h's and δ's. After applying *project&generalize*, we obtain for $h_\ell[i]$, two types of assertions. One is for the case that $i = z$, and then, as expected, $h_2[z]$ is the most interesting one, having an A-construct claiming that z's ticket is the minimal among ticket holders. The other case is for $j \neq z$, and there we have a similar A-construct (for j's ticket minimality) for $\ell = 2, 3, 4$. For the pre-order, one must consider $\tau_{\ell_1}[i] \preceq \tau_{\ell_2}[j]$ for every $\ell_1, \ell_2 = 1, ..., 4$ and $i = z \neq j, i = j \neq z, i, j \neq z$ for $(\ell_1, i) \neq (\ell_2, j)$. The results for $\tau_{\ell_1}[i] \preceq \tau_{\ell_2}[j]$ for $i \neq z$ that are not trivially T are described in Appendix 6.A.1

Using the above pre-order, we succeeded in validating Premises R1–R9 of PRERANK, thus establishing the liveness property of program BAKERY.

6.8 Multiple Pre-Order Relations

In the previous section we described how to compute the pre-order relation. Formula (6.2) is one alternative of computing the pre-order. We can view rule DISTRANK as a special case of rule PRERANK, with a trivial pre-order defined by $s \models \tau_1 \preceq \tau_2$ if $s \models \Psi(\tau_1, \tau_2)$, where

$$\Psi(\tau_1, \tau_2): \; h_{\tau_1} \; \vee \; \neg h_{\tau_2} \tag{6.3}$$

Obviously, other definitions are also possible. In fact, by allowing different schemes of computing pre-order on different states, the rule PRERANK can be applied to a wider range of protocols. In this section we demonstrate this idea on a version of Szymanski's mutual exclusion protocol described in Fig. 6.9.

The progress property we would ideally like to verify is $(\pi[z] = 1 \Rightarrow \Diamond(\pi[z] = 7)$. This property, however, is beyond the scope of the methods and rules described here since it requires some just transition to be helpful twice. It is not difficult, but rather tedious, to extend our technique for generating ranking so to deal with cases where transitions may be helpful up to k times, for any bounded k. We bypass this difficulty here by restricting to a "smaller" progress property to which the proof applies, namely, to the progress property

$$(\pi[z] = 1 \; \wedge \; \forall i : \pi[i] \leq 4) \; \Rightarrow \; \Diamond(\pi[z] = 7) \tag{6.4}$$

An inspection of the protocol reveals that $\tau_6[i]$ is the only transition whose enabling condition is of the form $\forall j.p(i, j)$ which is an obvious candidate for pre-ordering of the type we used in Section 6.7. The other transitions all have enabling conditions of the form $p(i) \; \wedge \; \forall j : q(j)$ (or simpler) that can be easily handled by the trivial pre-order which we implicitly use when applying DISTRANK. Consequently, we partition the concrete pending states into $pend_1 = \exists i :$

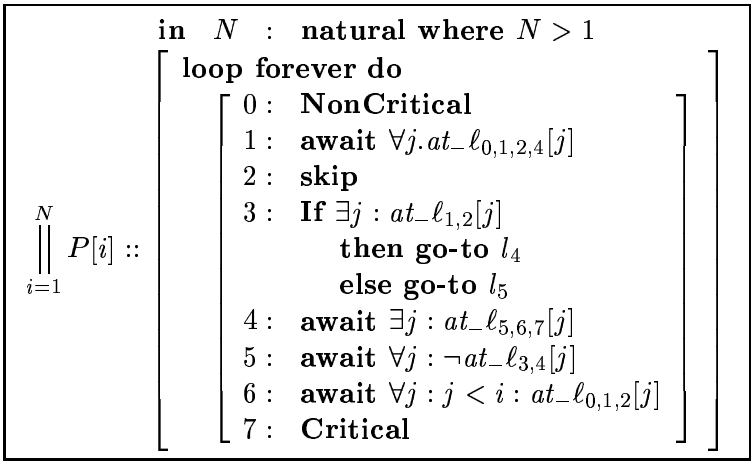


Figure 6.9: Program Szymanski

$\bigvee_{\ell \notin \{0,6\}} En(\tau_\ell[i])$ and $pend_2 = pend \wedge \neg pend_1$. The (concrete) pre-order is now defined for $pend_1$-states by

$$\tau_\ell[i] \preceq \tau_{\ell'}[i'] = \begin{cases} \Psi(\tau_\ell[i], \tau_{\ell'}[i']) & \ell, \ell' \neq 6 \\ \mathtt{T} & \ell' = 6 \\ \mathtt{F} & \text{otherwise} \end{cases}$$

and for $pend_2$-states by:

$$\tau_\ell[i] \preceq \tau_{\ell'}[i'] = \begin{cases} \Phi(\tau_\ell[i], \tau_{\ell'}[i']) & \ell = \ell' = 6 \\ \mathtt{T} & \ell' \neq 6 \\ \mathtt{F} & \text{otherwise} \end{cases}$$

where Ψ is defined in Formula (6.3) and Φ is defined Formula (6.2).

These definitions allow us to use *project&generalize* on the concrete pre-order (as described in Section 6.7) and successfully prove Formula (6.4) for program Szymanski.

Bibliography

[AK86] K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Info. Proc. Lett.*, 22(6), 1986.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *G. Berry, H. Comon, and A. Finkel, editors,* Proc. 13^{th} Intl. Conference on Computer Aided Verification (CAV'01), *volume 2102 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 221–234, 2001.

[BBC+95] N. Bjørner, I.A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR92)*, volume 962 of *Lect. Notes in Comp. Sci.*, pages 395–407, Philadelphia, PA, August 1995. Springer-Verlag.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comp. Systems Sci.*, 8:117–141, 1974.

[CS02] M. Colon and H. Sipma. Practical methods for proving program termination. In *E. Brinksma and K. G.Larsen, editors,* Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02), *volume 2404 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 442–454, 2002.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22nd ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[FPPZ04a] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with incomprehensible ranking. In *Proc.* 10^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), volume 2988 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 482–496, April 2004.

[FPPZ04b] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Proc. of the* 5^{th} *conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lect. Notes in Comp. Sci.*, pages 223–238, Venice, Italy, January 2004. Springer-Verlag.

[GS97] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *O. Grumberg, editor, Proc.* Proc. 9^{th} Intl. Conference on Computer Aided Verification, (CAV'97), *volume 1254 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 1997.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In *B. Steffen, editor, Proc.* 4^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 424–438, 1998.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *S. Graf and M. Schwartzbach, editors, Proc.* 6^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), volume 1785 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 2000.

[KPP03] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. In *W. Hunt Jr and F. Somenzi, editors* Proc. 15^{th} Intl. Conference on Computer Aided Verification (CAV'03), pages 381–392, Boulder, CO, USA, August 2003.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A.J. Hu and M.Y. Vardi, editors, *A.J. Hu and M.Y. Vardi, editors,* Proc. 10^{th} Intl. Conference on Computer Aided Verification (CAV'98), *volume 1427 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 110–121, 1998.

[MP91] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[OSR93] S. Owre, N. Shankar, and J.M. Rushby. User guide for the PVS specification and verification system (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.

$$
\begin{array}{ll}
V: & \left\{ \begin{array}{l} y: \ \mathbf{array}[1..N] \ \mathbf{of} \ [0..N] \\ \pi: \ \mathbf{array}[1..N] \ \mathbf{of} \ [0..4] \end{array} \right. \\
\Theta: & \forall i : \pi[i] = 0 \ \wedge \ y[i] = 0 \\
\mathcal{T}: & \left\{ \begin{array}{lll}
\tau_0(i): & \forall j \neq i: & \pi[i] = 0 \wedge \pi'[i] \in \{0,1\} \wedge \\
 & & pres(\pi[j], y[i], y[j]) \\
\tau_1(i): & \forall j,k \neq i: & \pi[i] = 1 \wedge \pi'[i] = 2 \wedge y'[j] < y'[i] \\
 & & \wedge \left(\begin{array}{c} y[j] = 0 \leftrightarrow y'[j] = 0 \wedge \\ y[j] < y[k] \leftrightarrow y'[j] < y'[k] \end{array} \right) \\
 & & \wedge pres(\pi[j]) \\
\tau_2(i): & \forall j \neq i: & \pi[i] = 2 \wedge (y[j] = 0 \vee y[j] > y[i]) \\
 & & \wedge \pi'[i] = 3 \wedge pres(\pi[j], y[i], y[j]) \\
\tau_3(i): & \forall j \neq i: & \pi[i] = 3 \wedge \pi'[i] = 4 \wedge \\
 & & pres(\pi[j], y[i], y[j]) \\
\tau_4(i): & \forall j \neq i: & \pi[i] = 4 \wedge \pi'[i] = 0 \wedge y'[i] = 0 \wedge \\
 & & pres(\pi[j], y[j]) \\
\tau_{id}: & \forall j: & pres(\pi[j], y[j])
\end{array} \right. \\
\mathcal{J}: & \{\tau_1(i), \tau_2(i), \tau_3(i), \tau_4(i) \ \mid \ i \in [1..N]\} \\
\mathcal{C}: & \emptyset
\end{array}
$$

Figure 6.10: BFTS for Program BAKERY

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. 7th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 82–97, 2001.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction, 2002.

[Sha00] E. Shahar. *The TLV Manual*, 2000. http://www.wisdom.weizmann.ac.il/~verify/tlv.

[Szy88] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, St. Malo, France, 1988.

[Var91] M. Y. Vardi. Verification of concurrent programs – the automata-theoretic framework. *Annals of Pure and Applied Logic*, 51:79–98, 1991.

[ZP04] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. *Computer Languages, Systems, and Structures*, 2004. to appear.

6.A BFTS's and Auxiliary Constructs

6.A.1 Program BAKERY

BFTS: See Fig. 6.10.

Auxiliary Constructs The auxiliary constructs for Program BAKERY with *minid* are:

$$\varphi_A: \quad \forall i: \quad \begin{pmatrix}(at\_\ell_{0,1}[i] \leftrightarrow y[i]=0) \wedge \\ (at\_\ell_{3,4}[i] \rightarrow \mathbf{minid}=i)\end{pmatrix} \wedge$$
$$\forall i{\neq}j: \quad \begin{pmatrix}(\mathbf{minid} \neq i \ \vee \ y[j]>y[i] \wedge y[i]\neq 0 \vee \\ y[j]=0) \wedge (y[i]=y[j] \rightarrow y[i]=0)\end{pmatrix}$$

$$pend_A: \quad \varphi_A \ \wedge \ at\_\ell_{1,2}[z]$$

$(h_\ell[j])_A$:

ℓ	For $j=z$	For $j\neq z$
1	$at\_\ell_1[z]$	0
2	$at\_\ell_2[z] \wedge$ $\mathbf{minid}=z$	$at\_\ell_2[z] \ \wedge \ at\_\ell_2[j]$ $\wedge \mathbf{minid}=j$
3	0	$at\_\ell_2[z] \ \wedge \ at\_\ell_3[j]$
4	0	$at\_\ell_2[z] \ \wedge \ at\_\ell_4[j]$

$(\delta_\ell[j])_A$:

ℓ	For $j=z$	For $j\neq z$
1	$at\_\ell_1[z]$	0
2	$at\_\ell_{1,2}[z]$	$\zeta(z,j,\{2\})$
3	0	$\zeta(z,j,\{2,3\})$
4	0	$\zeta(z,j,\{2,3,4\})$

where $\zeta(z,j,A) = at\_\ell_1[z] \ \vee \ at\_\ell_2[z] \ \wedge \ y[z]>y[j] \ \wedge \ at\_\ell_A[j]$.

Pre-order relation for non-*minid*-version Let α: $\pi[j]=2 \rightarrow y[z]<y[j]$, β: $\pi[i]=2 \wedge y[i]<y[j]$, and $\gamma(L)$: $\pi[j]\in L \rightarrow y[z]<y[j]$. The pre-order is described in Fig. 6.12.

6.A.2 Program TOKEN-RING

Symbolic Assertions

	$k=0$	$k=1$	$k=2$
$h_k^A[1]$	0	$at\_\ell_1[1] \ \wedge \ tloc=1$	0
$h_k^A[i], i>1$	$at\_\ell_1[1] \ \wedge \ at\_\ell_k[i] \ \wedge \ tloc=i$		

Symbolic Ranking

$$\left.\begin{array}{ll}
\delta_0^A[1]: & 0 \\
\delta_1^A[1]: & at_\ell_1[1] \\
\delta_2^A[1]: & 0 \\
\delta_0^A[i]: & at_\ell_1[1] \wedge \\ & (1<tloc<i \ \wedge \ at_\ell_{0,1}[i] \ \vee \ tloc=i) \\
\delta_1^A[i]: & at_\ell_1[1] \wedge \\ & \left(\begin{array}{l}1<tloc<i \ \wedge \ at_\ell_{0,1}[i] \vee \\ \quad tloc=i \ \wedge \ at_\ell_1[i])\end{array}\right) \\
\delta_2^A[i]: & at_\ell_1[1] \wedge \\ & \left(\begin{array}{l}1<tloc<i \ \wedge \ at_\ell_{0,1}[i] \vee \\ \quad tloc=i \ \wedge \ at_\ell_{1,2}[i]\end{array}\right)
\end{array}\right\} \quad \text{for } i>1$$

6.A.3 Program DINE

BFTS: See Fig. 6.11

Abstract Pending Sets

$$\varphi^A_{=1} = \left(\begin{array}{ll} & (y[N] \rightarrow \pi[N] < 2) \\ \wedge & (\pi[1] > 1 \rightarrow \pi[N] < 2) \\ \wedge & \left(y[1] \leftrightarrow \left(\begin{array}{ll} \pi[N] < 2 & \wedge \\ \pi[1] < 2 & \end{array} \right) \right) \end{array} \right)$$

$$\varphi^A_{\neq 1,N}(i, i{-}1) = \left(\begin{array}{ll} & (y[i{-}1] \rightarrow \pi[i{-}1] < 2) \\ \wedge & (\pi[i{-}1] > 2 \rightarrow \pi[i] < 2) \\ \wedge & \left(y[i] \leftrightarrow \left(\begin{array}{ll} \pi[i{-}1] < 3 & \wedge \\ \pi[i] < 2 & \end{array} \right) \right) \end{array} \right)$$

$$\varphi^A_{=N} = \left(\begin{array}{ll} & y[N-1] \rightarrow \pi[N-1] < 2 \\ \wedge & \pi[N-1] > 2 \rightarrow \pi[N] < 3 \\ \wedge & \left(y[N] \leftrightarrow \left(\begin{array}{ll} \pi[N-1] < 3 & \wedge \\ \pi[N] < 3 & \end{array} \right) \right) \end{array} \right)$$

Symbolic Ranking and Helpful Sets For every $j = z+1, \ldots, N{-}1$:

$$\begin{array}{ll} h^A_1[j]: & \text{F} \\ h^A_2[j]: & \pi[j{-}1] = 2 \ \wedge \ nvr_2[j{-}1] \ \wedge \\ & \pi[j] = 2 \ \wedge \ \neg nvr_2[j] \\ h^A_3[j]: & \pi[j{-}1] = 2 \ \wedge \ nvr_2[j{-}1] \ \wedge \\ & \pi[j] = 3 \ \wedge \ \neg y[i] \\ h^A_4[j]: & \pi[j{-}1] = 2 \ \wedge \ nvr_2[j{-}1] \ \wedge \\ & \pi[j] = 4 \ \wedge \ \neg y[i] \\ \delta^A_1[j]: & \text{T} \\ \delta^A_2[j]: & \neg nvr_2[j] \ \wedge \\ & (\pi[j{-}1] = 2 \ \wedge \ nvr_2[j{-}1] \rightarrow \pi[j] < 3) \\ \delta^A_3[j]: & \neg nvr_2[j] \ \wedge \\ & (\pi[j{-}1] = 2 \ \wedge \ nvr_2[j{-}1] \rightarrow \pi[j] < 4) \\ \delta^A_4[j]: & \text{T} \end{array}$$

$$V: \quad \left\{\begin{array}{ll} y, nvr_1, nvr_2: & \textbf{array } [1..N] \textbf{ of bool} \\ \pi: & \textbf{array } [1..N] \textbf{ of } [0..4] \end{array}\right\}$$

$$\Theta: \quad \forall i.\ (\pi[i] = 0 \ \wedge \ y[i]$$

$$\mathcal{T}: \quad \left\{\begin{array}{l}
\tau_0(i): \quad \forall j \neq i: \\
\quad \left(\begin{array}{ll} \pi[i] = 0 \wedge \pi'[i] \in \{0,1\} & \wedge \\ pres(y[i], nvr_1[i], nvr_2[i]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_1(i): \quad \forall j \notin \{i, i \oplus 1\}: \\
\quad \left(\begin{array}{ll} \pi[i] = 1 \wedge \pi'[i] = 2 \wedge pres(nvr_1[i], nvr_2[i]) & \wedge \\ (i < N \rightarrow (y[i] \wedge \neg y'[i] \wedge pres(y[i+1]))) & \wedge \\ (i = N \rightarrow (y[1] \wedge \neg y'[1] \wedge pres(y[N]))) & \wedge \\ pres(\pi[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j])) & \end{array}\right) \\
\qquad \vee \\
\quad \left(\begin{array}{ll} \neg nvr_1[i] \wedge nvr_1'[i] \wedge pres(\pi[i], y[i], nvr_2[i]) & \wedge \\ pres(\pi[i \oplus 1], y[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_2(i): \quad \forall j \notin \{i, i \oplus 1\}: \\
\quad \left(\begin{array}{ll} \pi[i] = 2 \wedge \pi'[i] = 3 \wedge pres(nvr_1[i], nvr_2[i]) & \wedge \\ (i < N \rightarrow (y[i+1] \wedge \neg y'[i+1] \wedge pres(y[i]))) & \wedge \\ (i = N \rightarrow (y[N] \wedge \neg y'[N] \wedge pres(y[1])) & \wedge \\ pres(\pi[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j])) & \end{array}\right) \\
\qquad \vee \\
\quad \left(\begin{array}{ll} \neg nvr_2[i] \wedge nvr_2'[i] \wedge pres(\pi[i], y[i], nvr_1[i]) & \wedge \\ pres(\pi[i \oplus 1], y[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_3(i): \quad \forall j \neq i: \\
\quad \left(\begin{array}{ll} \pi[i] = 3 \wedge \pi'[i] = 4 & \wedge \\ pres(y[i], nvr_1[i], nvr_2[i]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_4(i): \quad \forall j \notin \{i, i \oplus 1\}: \\
\quad \left(\begin{array}{ll} \pi[i] = 4 \wedge \pi'[i] = 0 \wedge pres(nvr_1[i], nvr_2[i]) & \wedge \\ y'[i] \wedge y'[i \oplus 1] & \wedge \\ pres(\pi[i \oplus 1], y[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_{id}: \quad \forall j: pres(\pi[j], y[j], nvr_1[j], nvr_2[j])
\end{array}\right.$$

$$\mathcal{J}: \quad \{\tau_1(i), \tau_2(i), \tau_3(i), \tau_4(i) \ \mid \ i \in [1..N]\}$$

Figure 6.11: BFTS for Program DINE

	$\tau_1[j]$	$\tau_2[j]$	$\tau_3[j]$	$\tau_4[j]$
$\tau_1[i]$	$i = j$ $\vee \quad j \neq z$ $\vee \quad \pi[z] = 2$	$j \neq z \wedge \pi[z] = 2 \wedge \alpha$ $\vee$ $i = j = z \wedge \pi[z] = 1$	$j = z$ $\vee \quad (\pi[z] = 2 \wedge \alpha$ $\wedge \pi[j] \neq 3)$	$j = z$ $\vee \quad \pi[z] = 2 \wedge \alpha$ $\wedge \pi[j] < 3$
$\tau_2[i]$	$j \neq z$ $\vee \quad \pi[z] = 2$	$i = j$ $\vee \quad \beta$ $\vee \quad \pi[j] \neq 2$ $\vee \quad j \neq z \wedge y[z] < y[j]$	$j = z \vee \pi[z] = 1$ $\vee \quad i = j \wedge \pi[j] \neq 3$ $\vee \quad i \neq j \wedge (\pi[j] \notin \{2, 3\} \vee$ $\beta \vee y[z] < y[j])$	$j = z \vee \pi[z] = 1$ $\vee \quad i = j \wedge \pi[j] < 3$ $\vee \quad i \neq j \wedge (\pi[j] < 2$ $\vee \quad \beta \quad \vee y[z] < y[j])$
$\tau_3[i]$	$j \neq z$ $\vee \quad \pi[z] = 2$	$\neg(i = j = z) \wedge$ $(\pi[z] = 1 \vee \beta$ $\vee \pi[i] = 3 \vee \alpha)$	$i = j \vee j = z$ $\vee \quad \beta \vee \pi[i] = 3$ $\vee \quad \gamma(2, 3)$	$(i = j \wedge \pi[i] = 2)$ $\vee \quad \beta \vee \pi[i] = 3$ $\vee \quad \gamma(2..4)$ $\vee \quad \pi[z] = 1 \vee j = z$
$\tau_4[i]$	$j \neq z$ $\vee \quad \pi[z] = 2$	$\neg(i = j = z) \wedge$ $(\pi[z] = 1 \vee \beta$ $\vee \pi[i] > 2 \vee \alpha)$	$j = z \vee \beta$ $\vee \quad i \neq j \wedge \pi[i] > 2$ $\vee \quad \gamma(2, 3)$	$i = j \vee j = z$ $\vee \quad \beta \vee \pi[i] > 2$ $\vee \quad \gamma(2..4)$

Figure 6.12: Pre-order for Program BAKERY

Chapter 7

Conclusions

We have considered verification of two types of infinite-state systems. For systems that use a pushdown store as a memory device we provide algorithms that extend the automata-theoretic approach for reasoning about infinite-state systems. We solve the linear-time model-checking problem with respect to pushdown and prefix-recognizable systems. We give the first solution for linear-time model checking with respect to prefix-recognizable systems. We establish that the problems of model-checking with respect to prefix-recognizable systems and to pushdown systems with regular labeling are intereducible. We extend the automata-theoretic framework to handle global model-checking for both branching-time and linear-time specifications. We consider the case where the specification is non-regular and show that we can model-check pushdown specifications with respect to finite-state systems but not with respect to context-free systems. We introduce the class of micro-macro stack graphs and offer model-checking solutions for both branching-time and linear-time specifications.

Our model-checking algorithms (both local and global) generalize to the logic CARET, which can specify non-regular properties [AEM04]. We believe that our global model-checking algorithms generalize also to micro-macro stack systems as well as high order pushdown systems [KNU03, Cac03].

Cachat et al. consider games over pushdown arenas where the winning condition is to visit some configuration infinitely often. They show that this winning condition is topologically more complex than the standard winning conditions normally used in such games [CDT02]. Walukiewicz et al. show that this result can be generalized to winning conditions that are Boolean combinations of the Büchi condition and the requirement that the size of the store be unbounded (termed *explosion condition*) [BSW03]. Gimbert extends this result to Boolean combinations of explosion condition and the more general parity condition [Gim03]. Recently, Serre showed games whose winning conditions are of increasing topological complexity that are decidable [Ser04]. We are working on extending the automata-theoretic framework to infinite-state systems to give a uniform solution for this type of games.

Since their introduction in [Cau96], prefix-recognizable graphs have been thoroughly studied. As a few examples we mention games on prefix-recognizable graphs [Cac02], characterization of languages accepted by prefix-recognizable graphs [Sti00], and prefix-recognizable structures [Blu01]. There are also many equivalent ways to represent prefix-recognizable graphs, using rewrite rules, as the outcome of regular restriction and inverse regular substitution on the infinite binary tree [Cau96], as monadic second-order logic interpretations in the infinite binary tree [Blu01], and as

graph equations [Cau96, Bar97]. All these issues need to be studied for micro-macro stack graphs.

The type of micro-macro partition of states with an ability to read the entire stack, can be applied also to high-order pushdown automata. Adding this ability to high-order pushdown automata of level i, produces graphs that are configuration graphs of high-order pushdown automata of level $i+1$ (but not all level $i+1$). Thus, by using this transformation we gain 'half a level'. Reasoning about such systems, however, is as complex as reasoning about the class of systems of level $n+1$. Further research is needed in order to understand exactly what is gained by using this type of transformation and whether the complexity involved in reasoning about micro-macro stack cannot be improved.

For parameterized systems, we provide a heuristic for proving liveness properties. Our heuristic is based on automatic generation of the auxiliary assertions needed by a deductive rule for proving liveness properties and proving the premises of the proof rule by finite state methods using a small (or modest) model theorem. We show how our framework can handle expressions containing ± 1 and universally quantified helpful assertions.

As mentioned, in some cases, universally quantified assertions are not sufficient to complete the deductive proof. For such cases, we show how to combine universally and existentially quantified assertions. By iterating the computation of universal and existential assertions we can get better approximations of the concrete sets we wish to capture. This process either reaches a fixpoint or captures exactly the concrete set. In both cases, further research is needed in order to be able to set the size of the instantiation on which the concrete set is computed so that this computation produces best results for all sizes of instantiation. In addition, we would like to continue applying this heuristic for other examples as well as to larger classes of systems.

Bibliography

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. 10th International Conference on Tools and Algorithms For the Construction and Analysis of Systems*, volume 2725 of *Lecture Notes in Computer Science*, pages 67–79, Barcelona, Spain, April 2004. Springer-Verlag.

[Bar97] K. Barthelmann. On equational simple graphs. Technical report, Universität Mainz, 1997.

[Blu01] Achim Blumensath. Prefix-recognisable graphs and monadic second-order logic. Technical Report AIB-06-2001, RWTH Aachen, May 2001.

[BSW03] A.-J. Bouquet, O. Serre, and I. Walukiewicz. Pushdown games with unboundedness and regular conditions. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2003.

[Cac02] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. In *4th International Workshop on Verification of Infinite-State Systems*, *Electronic Notes in Theoretical Computer Science* 68(6), Brno, Czech Republic, August 2002.

[Cac03] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *Proc. 30th International Colloqium on Automata, Languages, and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569, Eindhoven, The Netherlands, June 2003. Springer-Verlag.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 1996.

[CDT02] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a σ_3 winning condition. In *Proc. 11th Annual Conference of the European Association for Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 322–336. Springer-Verlag, 2002.

[Gim03] H. Gimbert. Explosion and parity games over context-free graphs. Technical Report 2003-015, Liafa, CNRS, Paris University 7, 2003.

[KNU03] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, Grenoble, France, April 2003. Springer-Verlag.

[Ser04] O. Serre. Games with winning conditions of high borel complexity. In *Proc. 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 1150–1162. Springer-Verlag, 2004.

[Sti00] C. Stirgling. Decidability of bisimulation equivalence for pushdown processes. Unpublished manuscript, 2000.