Modal Transition System Encoding of Featured Transition Systems

Mahsa Varshosaz[a], Lars Luthmann[b], Paul Mohr[b], Malte Lochau[b], Mohammad Reza Mousavi[c]

[a]*Halmstad University, Sweden*
[b]*TU Darmstadt, Germany*
[c]*University of Leicester, UK*

Abstract

Featured transition systems (FTSs) and modal transition systems (MTSs) are two of the most prominent and well-studied formalisms for modeling and analyzing behavioral variability as apparent in software product line engineering. On one hand, it is well-known that for finite behavior FTSs are strictly more expressive than MTSs, essentially due to the inability of MTSs to express logically constrained behavioral variability such as persistently exclusive behaviors. On the other hand, MTSs enjoy many desirable formal properties such as compositionality of semantic refinement and parallel composition. In order to finally consolidate the two formalisms for variability modeling, we establish a rigorous connection between FTSs and MTSs by means of an encoding of one FTS into an equivalent set of multiple MTSs. To this end, we split the structure of an FTS into several MTSs whenever it is necessary to denote exclusive choices that are not expressible in a single MTS. Moreover, extra care is taken when dealing with infinite behaviour: loops may have to be unrolled to accumulate FTS path constraints when encoding them into MTSs. We prove our encoding to be semantic-preserving (i.e., the resulting set of MTSs induces, up to bisimulation, the same set of derivable variants as their FTS counterpart) and to commute with modal refinement. We further give an algorithm to calculate a concise representation of a given FTS as a minimal set of MTSs. Finally, we present experimental results gained from applying a tool implementation of our approach to a collection of case studies.

Keywords: Featured Transition Systems, Modal Transition Systems, Expressiveness Power, Product Lines, Modeling

Email addresses: mahsa.varshosaz@hh.se (Mahsa Varshosaz), lars.luthmann@es.tu-darmstadt.de (Lars Luthmann), malte.lochau@es.tu-darmstadt.de (Malte Lochau), mm789@le.ac.uk (Mohammad Reza Mousavi)

1. Introduction

Different formal models have been proposed to capture the behavior of software product lines (SPLs), e.g., for model-based testing or model checking. Examples of such formal models include featured transition systems (FTSs) [1], modal transition systems (MTSs) [2] and various extensions thereof [3, 4, 5, 6, 7, 8, 9, 10, 11]. The expressive power of some of the aforementioned formalisms has been assessed in [12, 13, 14, 15]. The comparison of the expressiveness is established based on proving the (non-)existence of an encoding, which is a transformation from one class of models to the other by preserving the set of derivable model variants in terms of implementing Labeled Transition Systems (LTSs). (The provided results cover models with infinite state or finite behavior.) As a part of the results, it is shown that FTSs with finite behavior are more expressive than plain MTSs with finite behavior (i.e., MTS without any additional constructs to express variability constraints), essentially because those plain MTSs cannot specify persistently exclusive behavior. However, the theory of MTSs has been extensively studied [10] and based on that, various tools have been developed to support their analysis [5, 16, 17, 18, 19, 20]. In addition, MTSs enjoy many desirable formal properties such as inherent notions of semantic refinement being compatible with parallel composition, thus enabling compositional reasoning. Hence, it makes sense to further explore the connection between FTSs and MTSs and to come up with semantic-preserving encoding of FTSs into MTSs.

We address this problem by providing a transition of one FTS into *a set* of multiple MTSs. The MTSs considered in this work are congruent with the ones defined by Larsen et al. in [2], which extend LTSs by a may-/must-modality of single transitions. An alternative approach [21, 11, 13] is to encode FTSs into MTSs by annotating the target MTSs with variability constraints when needed. Our encoding only splits the structure of a given FTS into multiple mutually excluding MTSs when it is necessary, i.e., when there is an exclusive choice among transitions in the FTS that cannot be captured by one single MTS. We prove that our translation allows for step-wise refinement, i.e., it is consistent with the existing notions of refinement on MTSs and FTSs and it is semantic preserving, i.e., it induces, up to bisimulation, similar sets of products for the resulting set of MTSs as the original FTS. We also give an algorithm to calculate the translated MTSs and prove it correct with respect to our definition. A number of essential concerns are addressed in the definition of this encoding: firstly, the path constraints accumulated through different paths may turn out to be inconsistent with each other and hence, such paths have to be split into different MTSs. Moreover, paths are accumulated and potentially strengthened through loops and hence, loops may have to be unrolled to cater for this. We further consider the issue of minimality of FTS encodings into sets of MTSs and show that our proposed algorithm satisfies this notion for structurally deterministic FTSs.

In addition, we present experimental evaluation results gained from applying a tool implementation of our approach to a collection of case studies on FTSs [22] as well as from a collection of synthetically generated, yet realistic FTS models. The goal of this empirical study is to show scalability of our tool also to larger models and to investigate efficiency and effectiveness of generating a minimal MTS encoding from FTS input models. In particular, we investigate the computational effort for generating

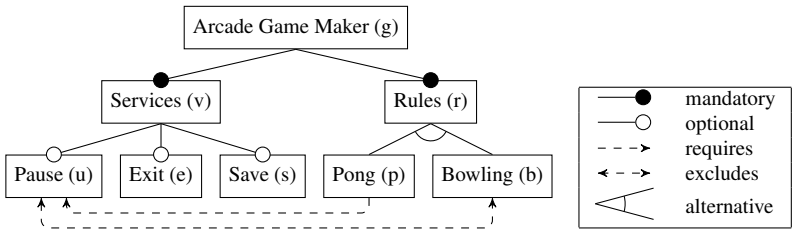


Figure 1: The Feature Model of the Arcade Game Maker Product Line

MTS models from a given FTS as well as the average number of MTSs as compared to the overall number of variants.

The rest of this paper is organized as follows. In Section 2, we introduce a running example, namely the Arcade Game Maker product line [23, 24], that is used to illustrate the concepts and notions. In Section 3, we formally introduce the basic concepts regarding labeled transition systems (LTSs), MTSs, and FTSs. In Section 4, a valid encoding of FTSs into sets of MTSs is characterized in a declarative way and is shown to be semantic preserving. In Section 5, an algorithmic view of the encoding is provided and its correctness is proven. Moreover, a notion of minimality is proposed in the same section and the outcome of the algorithm is shown to satisfy this notion for FTSs that are structurally deterministic. In Section 6, we evaluate our algorithm on a number of (real-world as well as synthetic) case studies. In Section 7, an overview of the literature in this area is given and different pieces of the literature are related to the present work. Finally, in Section 8, the paper is concluded and some avenues for future research are discussed.

2. Running Example

In this section, we first introduce an illustrative example, a simple software product line of an *Arcade Game Maker (AGM)*, which will be used throughout this paper. The AGM product line comprises two different games (*rules*), namely *pong* and *bowling*. In addition, the AGM application enables the player to use different *services* such as *pausing* and *exiting* a running game as well as *saving* the recent game.

A common notation for a compact representation of the set of features and the relations between them are *feature models*, usually visualized in terms of feature diagrams [25]. A feature diagram for our AGM example is depicted in Figure 1. A feature diagram is a tree-like structure in which each node represents a feature. Each single feature is either *mandatory*, if it is included in all the products of the product line in which its parent feature is included, or it is *optional*, otherwise. In addition, a feature can have *groups* of sub-features of two different kinds. First, a set of sibling features can be in an *or*-relation, which means that *at least one* of the features has to be selected whenever the parent feature of that group is selected (not contained in our example). Second, a set of sibling features can be in an *xor*-relation which means that *exactly one* of the features has to be selected whenever the parent feature is selected (alternative group in our example). Finally, a feature may be in a *require-* or

(mutual) *exclusion*-relation with another feature, represented by a (respectively, solid and dashed) *cross-tree edge*. Concerning the feature model in Figure 1, the diagram contains compound features for the configuration of *rules* and *services*. Features *pong* and *bowling* have an xor-relationship, whereas the service features are all optional. In addition, feature *pong* requires feature *pause*, whereas feature *bowling* excludes feature *pause*. A *valid configuration* of a product line corresponds to a subset of features satisfying all the constraints of the feature model. For example, the AGM feature model in Figure 1 has 8 valid configurations.

3. Foundations

In this section, we explain the constructs and concepts used throughout this paper. In particular, we consider two existing formalisms for product-line modeling namely, MTSs and FTSs. Each abstract model belonging to one of these two classes of models comprises several concrete *implementation variants* in terms of labeled transition systems (LTSs). The notion of LTS is defined as follows (cf. [26]).

Definition 1 (Labeled Transition System)**.** *A labeled transition system is a tuple* $(S, A, \rightarrow, s_{init})$*, where:*

- S *is a finite set of* states*,*
- A *is a finite set of* actions*,*
- $\rightarrow \subseteq S \times A \times S$ *is a (labeled)* transition relation*,*
- $s_{init} \in S$ *is an* initial state*.*

As an example, consider the LTS in Figure 2a depicting a configuration of the AGM. A similar, yet slightly different LTS model can be given for each of the 7 other valid product configurations of the AMG product line, each sharing certain common behaviors and differing in variable behaviors.

An LTS-based formalism considered for expressing behavioral commonality and variability in a product line are modal transition systems (MTSs), as shown in Figure 2b for our AGM example. In an MTS model, the set of transitions is subdivided into subsets of *must*-transitions denoting *mandatory* (core) behavior to be included in every configuration, and *may*-transitions denoting *optional* behaviors. Note that every must-transition also requires an "underlying" may-transition as a must-transition always needs to be allowed as well. Hence, we call must-transitions with an underlying may-transition *mandatory* transitions, whereas may-transitions with no corresponding must-transition are called *optional* transitions. In figures, we use solid lines to denote mandatory transitions and dashed lines to denote optional transitions.

(The MTSs that we consider in this work are complying with the original definition given by Larsen et al. in [2]. Different extensions of MTSs have been provided e.g., MTSs with variability constraints [11] and parametric MTSs [27] that provide means to explicitly relate features to the behavior similar to FTSs. However, these extensions of MTSs are out of the scope of this preliminary work of MTS encodings of FTSs.)

An MTS may be formally defined as follows (cf. [2]).

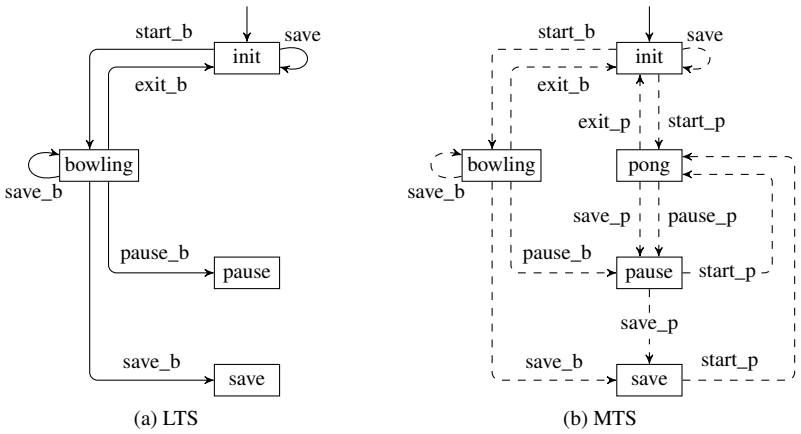


Figure 2: LTS and MTS for the AGM Example

Definition 2 (Modal Transition System)**.** *A modal transition system is a 5-tuple* $(S, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, s_{init})$ *where:*

- S *is a finite set of* states*,*
- A *is a finite set of* actions*,*
- $\rightarrow_{\Diamond} \subseteq S \times A \times S$ *is a* may-transition relation*,*
- $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond}$ *is a* must-transition relation*,*
- $s_{init} \in S$ *is an* initial state*.*

Hence, those transitions being contained in the set of *may*-transitions but not in the set of *must*-transitions express optional (or *variable*) behaviors of an SPL. Hence, an MTS integrates a *set* of LTSs which can be obtained via *modal refinement* (i.e., every optional transition either becomes a mandatory transition, or it is removed from the model). In this regard, an LTS may be considered as an MTS in which $\rightarrow_{\Diamond} = \rightarrow_{\Box}$ holds.

In order to formally define the set of valid implementations of an MTS, we employ the (modal) refinement relation for MTS, based on Larsen et al. [2], as follows.

Definition 3. *Consider two MTSs,* $mts' = (S, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, s_{init})$ *and* $mts = (T, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, t_{init})$*. A binary relation* $\mathcal{R} \subseteq S \times T$ *is a* modal refinement relation *if and only if the following properties are satisfied.*

1. $\forall_{t,t' \in T, s \in S, a \in A} \big(s \mathcal{R} t \wedge t \xrightarrow{a}_{\Box} t' \big) \implies \exists_{s' \in S}\ s \xrightarrow{a}_{\Box} s' \wedge s' \mathcal{R} t'$*, and*
2. $\forall_{s,s' \in S, t \in T, a \in A} \big(s \mathcal{R} t \wedge s \xrightarrow{a}_{\Diamond} s' \big) \implies \exists_{t' \in T}\ t \xrightarrow{a}_{\Diamond} t' \wedge s' \mathcal{R} t'$*.*

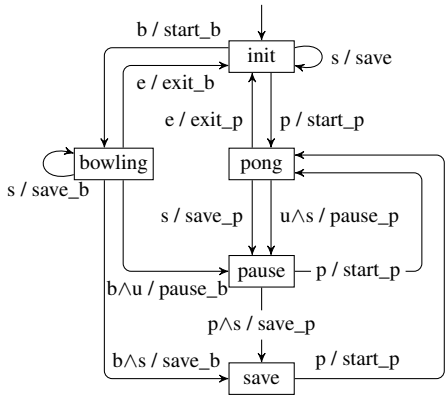


Figure 3: The Featured Transition System of the AGM Product Line

The modal specification mts′ refines *the modal specification mts, denoted mts′ $\preceq$ mts, if there exists a modal refinement relation $\mathcal{R}$ such that $s_{init}\mathcal{R}t_{init}$. We denote all the MTSs that refine the MTS M by $[\![M]\!]$.*

For each of the eight valid product configurations of the AMG product line, a corresponding LTS model can be derived from the MTS example in Figure 2b via modal refinement. For instance, the LTS in Figure 2a depicts a product where all optional transitions related to bowling become mandatory and all other optional transitions are removed. However, the converse statement does not hold as, in addition to those 8 valid LTS variants, further LTS variants may be derived from the MTS in Figure 2b that do *not* correspond to any valid configuration of the AGM product line. For instance, both the behaviors for feature p *and* feature b may be either preserved or removed under modal refinement which clearly contradicts the exclusive-or dependency among p and b as stated in the feature model in Figure 1. This example illustrates the inherent inability of MTS to express persistently exclusive choices among variable behaviors.

Another LTS-based formalism for expressing behavioral commonality and variability in a product line are featured transition systems (FTSs), as shown in Figure 3 for our AGM example. Similar to an LTS or MTS, an FTS consists of a set of states and a set of transitions, labeled with actions. In addition to actions, transitions of an FTS are further labeled with *presence conditions* over (Boolean) feature variables. The presence conditions determine those product configurations in which the transition in hand is included. In this way, an FTS incorporates an explicit notion of behavioral variability by virtually integrating a set of similar, yet well-distinguished LTS models into one product-line model.

The transition labels in the FTS in Figure 3 for the AGM product line are of the form "*presence condition / action*". In particular, the atomic proposition in the presence conditions refer to the (abbreviated) feature names in the feature model in Figure 1. If residing in initial state *init*, the AGM either enters a new game *bowling*, or a new game *pong*, respectively, whenever action *start* occurs. If the user triggers action *pause*, both types of games may be suspended by entering a *pause* state. In this particular example,

a *pong* game may be re-entered again via action *start*, which is, however, not supported in case of a *bowling* game. Instead, a *bowling* game may be *saved* during the game, whereas a *pong* game has to be *paused* before it can be *saved*. In addition, both kinds of games may be stopped by the *exit* action which leads the FTS back to the *init* state. For instance, choosing features $\neg u$, e, s, $\neg p$, and b results in the LTS being depicted in Figure 2a.

As described above, feature diagrams are frequently used for representing the set of features of a product line and the relations between them. Alternatively, the configuration constraints as (graphically) imposed in a feature diagram may be also represented as propositional formula over features, represented as Boolean variables. By $\mathbb{B}(F)$, we denote the set of propositional formulae over a set F of (Boolean) feature variables. We now give the formal definition of FTS based on [1], as follows.

Definition 4 (Featured Transition System)**.** *A featured transition system is a 6-tuple* $(S, A, F, \rightarrow, \Lambda, p_{init})$*, where*

- S *is a finite set of* states*,*
- A *is a finite set of* actions*,*
- F *is a finite set of* features*,*
- $\rightarrow \subseteq S \times \mathbb{B}(F) \times A \times S$ *is a* transition relation *satisfying the following condition:*

$$\forall_{S,a,S',\phi,\phi'} \left((S, \phi, a, S') \in \rightarrow \ \wedge (S, \phi', a, S') \in \rightarrow \right) \implies \phi = \phi',$$

- $\Lambda \subseteq \{\lambda : F \rightarrow \mathbb{B}\}$ *is a set of* product configurations*, and*
- $s_{init} \in S$ *is an* initial state*.*

In order to define the set of valid implementations of an FTS, we first give the following auxiliary definition.

Definition 5. *Considering a set of feature variables F and a set of product configurations Λ; for a propositional formula $e \in \mathbb{B}(F)$, we say $Sat(e)$, iff $\bigvee_{\lambda \in \Lambda} \lambda \ \wedge e$ is satisfiable.*

Next, we define a *product derivation relation* [12], that is used for extracting the set of valid implementations (or, LTS variants) of an FTS, as follows.

Definition 6. *Given an FTS $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda)$, and LTS $l = (\mathbb{S}, A, \rightarrow, s_{init})$, and a product $\lambda \in \Lambda$. A binary relation $\mathcal{R}_\lambda \subseteq \mathbb{P} \times \mathbb{S}$ (parameterized by product configurations) are called* product-derivation relations *if and only if the following transfer properties are satisfied.*

1. $\forall_{P,Q,a,s,\phi} \left(P \, \mathcal{R}_\lambda \, s \ \wedge \ P \xrightarrow{\phi/a} Q \wedge \lambda \models \phi \right) \ \Rightarrow \ \exists_t \cdot \ s \xrightarrow{a} t \ \wedge \ Q \, \mathcal{R}_\lambda \, t$*;*
2. $\forall_{P,a,s,t} \left(P \, \mathcal{R}_\lambda \, s \ \wedge \ s \xrightarrow{a} t \right) \ \Rightarrow \ \exists_{Q,\phi} \cdot \ P \xrightarrow{\phi/a} Q \ \wedge \ \lambda \models \phi \wedge Q \, \mathcal{R}_\lambda \, t.$

A state $s \in \mathbb{S}$ derives the product λ from an FTS-specification $P \in \mathbb{P}$, denoted by $P \vdash_\lambda s$, if there exists a product-derivation relation $\mathcal{R}_\lambda$ such that $P \, \mathcal{R}_\lambda \, s$.

We say that l is a valid implementation of fts, denoted by $fts \triangleright l$ if and only if there exists a product configuration $\lambda \in \Lambda$ such that $p_{init} \vdash_{\lambda} s_{init}$ holds. We denote all LTSs being derivable from the FTS fts by $[\![fts]\!]$.

Please note that Classen et al. in [1] provide a different "projection" operator for deriving the individual product models from an FTS. Based on their definition, an FTS is projected onto a product configuration, and as the result of projection, those transitions of the FTS for which the corresponding feature expression satisfies the product configuration are included in the product model whereas the other transitions are eliminated. This definition provides a syntactical description for deriving product models while the product-derivation relation given in Definition 6 constitutes a semantical notion of product-model derivation similar to modal refinement of MTSs. The sets of LTSs derived from an FTS (here for finite behavior) using either of these definitions are equal modulo bisimilarity (see Theorem 4 and its proof in Appendix B.). In this work, we use Definition 6 due to its declarative nature; for example, it allows for implementations that reduce the number of states while constructing the LTSs. It is also more suitable for providing the foundation for our encoding of FTSs into MTSs and more specifically for constructing the proofs to show that the encoding is semantic preserving.

For each of the eight valid product configurations of the AMG product line, a corresponding LTS model can be derived from the FTS model by deleting those transitions whose presence conditions are not satisfied by the corresponding product configuration (and by omitting the presence conditions of the remaining transitions).

It has been proven in recent literature [12], that FTSs with finite behavior are strictly more expressive than MTSs with finite behavior. More precisely, the comparison of the expressive power is based on the (non-)existence of a one-to-one encoding from one class of models into the other. In particular, such an encoding should define a translation of one model into another model having equal (modulo bisimilarity) sets of implementing LTSs. As illustrated by our example, there exist FTSs for which no *single* MTS can induce the same set of LTS models as valid product-line configurations [12]. However, if we consider *multiple* MTS models to characterize sets of valid LTSs corresponding to an FTS, then every FTS is expressible in terms of (a set of) MTSs. A general result about this relationship will be constructively proven in the remainder of this paper.

As an alternative line of work, we refer to the extension of MTSs with feature constraints [21, 13]; in this line of work, the authors provide a translation from FTSs to MTSs by annotating the target MTSs with feature constraints when necessary.

4. From FTS to MTSs

The goal of this section is to define a semantic preserving translation, called an encoding. We first set the scene by motivating the basic concepts used in our encoding from an FTS to a set of MTSs. Subsequently, we define our encoding and prove its correctness.

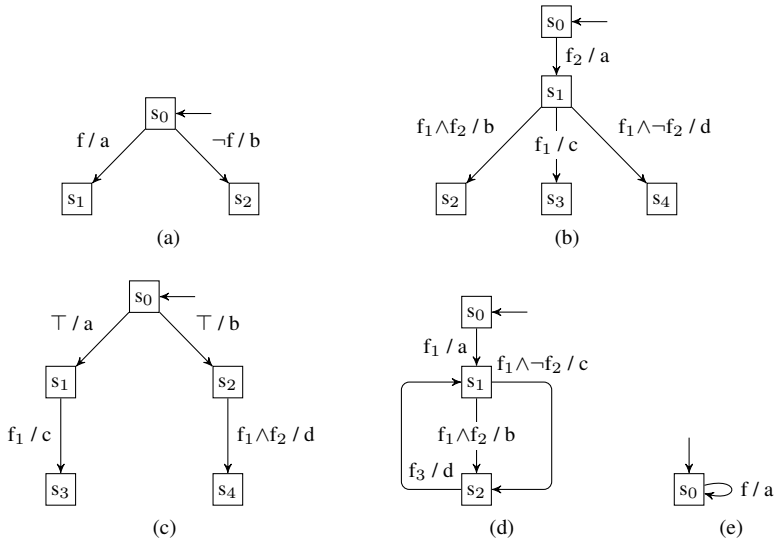


Figure 4: Four Example FTSs used to Motivate Various Concepts in the FTS Encoding

4.1. Encoding Concepts

As stated before, MTSs are inherently incapable of capturing mutually exclusive behavior that is naturally expressible in FTSs. To illustrate this in terms of a minimal example, consider the FTS in Figure 4a; there is an excludes dependency between the two transitions emanating from initial state s_0. Assume towards a contradiction that an MTS could model the behavior of the same product line. Then, the initial state of the purported MTS must include both an a-labeled and a b-labeled outgoing may-transition (otherwise, it fails to produce one of the LTSs, either having an outgoing a- or an outgoing b-labeled transition). However, in such a case, there is an LTS product of the purported MTS that has both outgoing a- and b-labeled transitions from the initial state, which is not a valid product of the FTS.

Hence, whenever the presence conditions of the emanating may- or must-transitions are not consistent in the FTS (i.e., there are dependencies such as excludes or requires relations between presence conditions of transitions in the FTS), we have to split the MTS structure into maximal subsets of transitions without such conflicts, thus leading to a set of MTSs for a given FTS.

For instance, for the FTS in Figure 4a, this leads to the two MTSs as depicted in Figure 5a.

However, consistency of transitions in one MTS is not only dependent on their presence conditions as stated in the FTS, but also on the *path conditions* accumulated from the presence conditions of other transitions traversed on the different possible paths reaching the source state of the transition under consideration.

For example, consider the FTS in Figure 4b: state s_1 has three outgoing transitions of which the presence conditions appear to be inconsistent at first sight. However, state s_1 is only reachable from the initial state through the a-labeled transition having

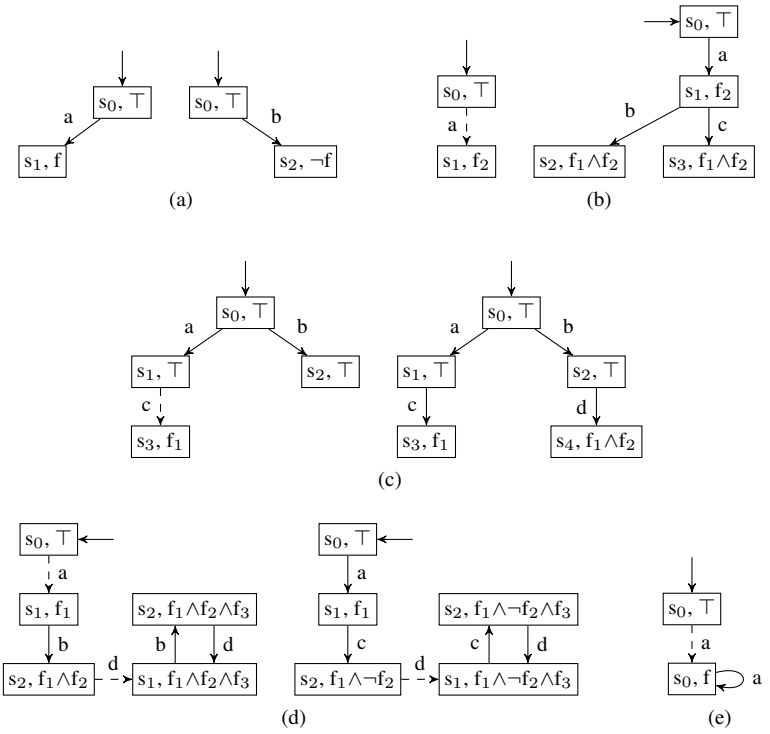
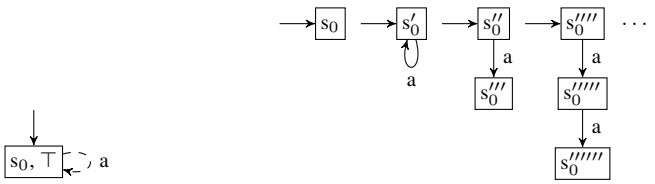


Figure 5: MTS Encodings of the FTSs Depicted in Figure 4

presence condition f_2. Hence, when arriving at state s_1, the path condition f_2 must hold and hence, the outgoing d-labeled transition is not present. The remaining two outgoing transitions are mutually dependent such that all derivable LTSs containing the b-labeled transition with presence condition $f_1 \wedge f_2$ must also include the c-labeled transition. Conversely, if the c-labeled transition is present in an LTS, then both features f_1 and f_2 are present in the product configuration and hence the b-labeled transition must also be included in the LTS. Hence, we require two MTSs to interpret the respective FTS, which are shown in Figure 5b: one representing the behavior of products that include feature f_1 and the other representing the behavior of the remaining products.

To generalize, not only consistency of the path conditions of transitions leaving the *same* state must hold as illustrated in the previous example, but rather consistency of path conditions of *all* transitions in one MTS must hold. Figure 4c represents an FTS in which there is a configuration dependency between the c-labeled and d-labeled transitions. This dependency is similar to the one between the transitions in Figure 4b, but the concerned transitions are now located on different paths. Hence, we again require two MTSs to interpret the FTS, as shown in Figure 5c.

Finally, special care is required for handling loops in the state-transition graph of FTS in the respective MTS encodings. Here, loops may have to be unrolled to a certain depth in order to correctly encode the accumulated path constraints for the transitions



(a) mts_i (b) Variants of mts_i

Figure 6: Example for the Necessity of Loop Unrolling

involved. For instance, consider the FTS depicted in Figure 4d: for reaching state s_2, the path condition $f_1 \wedge f_2$ or $f_1 \wedge \neg f_2$ must hold. When going from here back to state s_1, the path constraint is stricter now than when we visited s_1 for reaching s_2 for the first time. This is reflected in the MTSs depicted in Figure 5d, where the loop has to be unrolled once to distinguish the different path conditions. The intuition behind this is that if the optional transition labeled with d is included in a variant derived from the MTS on the left via refinement, then this action must always be enabled, again, afterwards whenever reaching some state related to FTS state s_2 in the same variant (as enforced by the presence condition in corresponding FTS). This can be encoded into MTSs only by unrolling the respective transition loop such that the first occurrence of an action is attached to an optional transition, whereas all subsequent occurrence(s) are attached to mandatory transitions. Figures 4e, 5e, and 6 provide another example, where we assume feature f to be optional. Hence, the FTS in Figure 4e has exactly two variants: one without any actions and a second one in which action a may be performed arbitrarily often. These are also exactly the variants derivable from MTS mts_c (cf. Figure 5e). Additionally, at a first glance, mts_i as depicted in Figure 6a seems to be a smaller MTS (in the number of states and transitions), yet having the same variants. However, in contrast to *fts* and mts_c, due to modal refinement, MTS mts_i further comprises an infinite number of different variants, each permitting action a to be performed *at most* k times, with $k \in \mathbb{N}$, which is clearly not permitted by *fts*.

Given these basic cases, we now formally characterize MTS encodings of FTSs in a declarative way. We therefore introduce the notion of *context* of an FTS to contain those valid sets of MTSs having the same (union of) of sets of LTS implementations as the given FTS. Please note that the context of an FTS is not necessarily unique, i.e., there may be multiple valid sets of MTSs which represent the same behavior as an FTS.

To define the notion of context for FTSs, we first need to specify the set of valid products that a set of MTSs can specify. The states of MTS in the context of an FTS consist of pairs of states of the respective FTS together with a propositional formula (up to logical equivalence) denoting the path condition for reaching this state in the FTS. Based on this additional information, we are able to define the set of product configurations corresponding to the set of products implemented by an MTS by means of the set of FTS implementations implying the resulting propositional formula. The overall propositional formula for the whole MTS with respect to the given FTS is constructed using a recursive function defined using a fixed point construction, named *context*, as follows.

Starting from the initial state of the MTSs at hand, we assume that the set of outgoing transitions from the corresponding state; in the FTS; is partitioned into three different sets of transitions, namely must-, may- and excluded-transitions. Considering must-transitions (i.e., transitions being present in all valid implementations of the FTS), we build the conjunction of the path condition of the current state and the resulting propositional formula for the target state of the transition (being computed by recursively applying the *context*-function to that state). Since must transitions are present in all considered valid products, the product configurations corresponding to implemented products imply this conjunction. Instead, may-transitions (i.e., transitions present in some but not all valid implementations of the FTS), are represented by disjunction of the negated path condition of the current state the resulting propositional formula for the target state as described for the must-case. Finally, considering excluded-transitions (i.e., transitions inconsistent with the included transitions of the FTS), we build the conjunction of the negation of the presence conditions.

The *MTS constraint* given as the conjunction of the formulas constructed for all three sets therefore characterizes the set of LTS subsumed by the current MTS such that all product configurations implying this constraint correspond to products of the FTS implemented by this MTS.

Definition 7 (MTS Constraint)**.** *Consider an FTS fts* $=(\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$*, and an MTS* $mts=(Q, A, \rightarrow_\Diamond, \rightarrow_\Box, q_{init})$*, where* $Q = \mathbb{P} \times \mathbb{B}(F)$*. We define the corresponding* MTS constraint *as follows. We first define for each state* $(p, e) \in Q$ *the following notations:*

- $exc((p, e)) = \{(p, f, a, p') \in \rightarrow \mid \exists_{(p,e)\in Q}\ Sat(e \wedge f) \wedge ((p, e), a, (p', e \wedge f)) \notin \rightarrow_\Diamond\}$,
- $must((p, e)) = \{(p, a, f, p') \in \rightarrow \mid \exists((p, e), a, (p, e \wedge f)) \in \rightarrow_\Box\}$,
- $may((p, e)) = \{(p, a, f, p') \in \rightarrow \mid \exists((p, e), a, (p, e \wedge f)) \in \rightarrow_\Diamond\}$.

By $const(q_{init})$ *we denote the MTS constraint for* mts*, where for each* $(p, e) \in Q$*,* $const((p, e))$*, is the maximal fixed point (w.r.t. logical implication ordering) for the following function:*

$$const_i((p, e)) = e \wedge \bigwedge_{(p,a,f,p')\in must((p,e))} \big(const_{i-1}((p', e \wedge f))\big) \wedge \bigwedge_{(p,a,f,p')\in may((p,e))} \big(\neg f \vee (const_{i-1}((p', e \wedge f)))\big) \wedge \bigwedge_{(p,a,f,p')\in exc((p,e))} \neg f$$

where $\forall_{(p,e)\in Q}\ const_0(p, e) = e$*. Furthermore, we say* Λ_{mts} *denotes the set of product configurations corresponding to the products implementing* mts*, which is* $\Lambda_{mts} = \{\lambda \in \Lambda \mid \lambda \implies const(q_{init})\}$.

As a property of the function $const()$, based on the following lemma we prove that this function is monotone and hence always has a fixed point.

Lemma 1. *Considering the definition of the function $const()$, given in Definition 7, this function always has a maximal fixed point.*

Proof. The proof is included in the appendix. □

Next, we give the definition of a *consistent MTS* with respect to a given FTS.

Definition 8 (Consistent MTS)**.** *Given an FTS fts $=(\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$, an MTS $mts=(Q, A, \rightarrow_\Diamond, \rightarrow_\Box, q_{init})$ is a* consistent MTS *with respect to fts, iff the following properties hold:*

1. *$Q \subseteq \mathbb{P} \times \mathbb{B}(F)$ is a set of states s.t.:*
   $\forall_{p' \in P, e' \in B(F)}\ (p', e') \in Q$ iff $\exists_{(p,e)\in Q}\ \exists_{(p,l,f,p')\in\rightarrow}(e' = e \wedge f)$.
   Here, we only consider the set of states that are reachable from the initial state.
2. *A is a set of actions.*
3. *$q_{init} = (p_{init}, \bigvee_{\lambda\in\Lambda} \lambda) \in Q$ is the initial state.*
4. *$\rightarrow_\Box \subseteq \rightarrow_\Diamond \subseteq\ Q \times A \times Q$, where $\rightarrow_\Box$ and $\rightarrow_\Diamond$ are maximal sets satisfying the following properties.*

   (a) $\forall_{((p,e),l,(p',e'))\in\rightarrow_\Diamond}\ \exists_{(p,f,l,p')\in\rightarrow}\ e' = e \wedge f$

   (b) $\forall_{(p,f,l,p')\in\rightarrow}\ \forall_{(p,e)\in Q}\ \forall_{\lambda\in\Lambda_{mts}}\ (\lambda \models e \implies \lambda \models f) \Leftrightarrow (p, e) \xrightarrow{l} (p', e \wedge f) \in \rightarrow_\Box$

   (c) Considering any subset of may-transitions T such that $\rightarrow_\Box \subseteq T \subseteq \rightarrow_\Diamond$, it holds that:

$$\exists \lambda \in \Lambda_{mts} : \lambda \models \bigwedge_{((p_0,e),l,(p'_0,e\wedge f))\in T} f \wedge \neg \left(\bigwedge_{((p_1,e),l,(p'_1,e\wedge g))\notin T} g \right)$$

   Furthermore, for each $\lambda \in \Lambda_{mts}$, a set of transitions $T \neq \emptyset$ with the property stated above exists.

Given the definition of a consistent MTS with regard to an FTS, we define the set of conditions that a set of MTSs must satisfy in order to be a valid part of the *FTS context* of a given FTS.

Definition 9 (FTS Context)**.** *Given an FTS fts $=(\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$, a set of MTSs $\mathcal{M} = \bigcup_{1\leq i\leq n} mts^i$, where $mts^i =(Q^i, A^i, \rightarrow^i_\Diamond, \rightarrow^i_\Box, q^i_{init})$ is in the* context *of fts, denoted by $\mathcal{M} \in context(fts)$ iff all the MTSs in $\mathcal{M}$ are consistent according to Definition 8, and the following conditions hold.*

1. $\Lambda = \bigcup_{mts \in \mathcal{M}} \Lambda_{mts}$
2. $\forall_{(p,f,l,p') \in \rightarrow} \exists_{((p,e),l,(p',e')) \in \bigcup_{1 \leq i \leq n} \rightarrow^i_\Diamond} \; e' = e \wedge f$

In the above definition, the first condition indicates that the union of all products implementing at least one MTS in the considered set of MTSs must be equal to the set of products of the FTS. The second condition indicates that each transition in the FTS must be included in at least one MTS in the set of MTSs.

As an example, consider the MTSs mts and mts', respectively, on the left- and right-side in Figure 5d. First, Λ_{mts} is computed as described above. Assume the set of states in these MTSs are $Q = \{q_0 = (s_0, \top), q_1 = (s_1, f_1), q_2 = (s_2, f_1 \wedge f_2), q_3 = (s_1, f_1 \wedge f_2 \wedge f_3), q_4 = (s_2, f_1 \wedge f_2 \wedge f_3)\}$. Then, in general we have:

$$const_i(q_0) = \top \wedge \big(\neg(f_1) \vee (const_{i-1}((s_1, f_1)))\big)$$

Then, we calculate $const_{i-1}((s_1, f_1))$:

$$const_{i-1}((s_1, f_1)) = f_1 \; \wedge \; \big(const_{i-2}((s_2, f_1 \wedge f_2))\big) \; \wedge \big(\neg(f_1 \wedge \neg f_2)\big)$$

Next, we compute $const_{i-2}((s_2, f_1 \wedge f_2))$:

$$const_{i-2}((s_2, f_1 \wedge f_2)) = \big(f_1 \wedge f_2\big) \; \wedge \; \big(\neg f_3 \vee const_{i-3}((s_1, f_1 \wedge f_2 \wedge f_3))\big)$$

Then, $const_{i-3}((s_1, f_1 \wedge f_2 \wedge f_3))$ is computed:

$$const_{i-3}((s_1, f_1 \wedge f_2 \wedge f_3)) = \big(f_1 \wedge f_2 \wedge f_3\big) \wedge \big(const_{i-4}((s_2, f_1 \wedge f_2 \wedge f_3))\big) \wedge \neg(f_1 \wedge \neg f_2)$$

In the next step, $const_{i-4}((s_2, f_1 \wedge f_2 \wedge f_3))$ is computed as:

$$const_{i-4}((s_2, f_1 \wedge f_2 \wedge f_3) = \big(f_1 \wedge f_2 \wedge f_3\big) \; \wedge \; \big(const_{i-5}((s_1, f_1 \wedge f_2 \wedge f_3))\big)$$

Hence, considering the calculations, in the first step we have:

$$const_0(q_0) = \top, \; const_0(q_1) = f_1, \; const_0(q_2) = f_1 \wedge f_2$$
$$const_0(q_3) = f_1 \wedge f_2 \wedge f_3, \; const_0(q_4) = f_1 \wedge f_2 \wedge f_3$$

We include a part of the next iterations that are relevant to obtaining the final results:

$const_1(q_4) = f_1 \wedge f_2 \wedge f_3 \wedge \big(const_0(q_3)\big) = f_1 \wedge f_2 \wedge f_3 = const_2(q_4)$
$const_1(q_3) = f_1 \wedge f_2 \wedge f_3 \wedge \big(const_0(q_4)\big) \wedge \neg(f_1 \wedge \neg f_2) = f_1 \wedge f_2 \wedge f_3$
$= const_2(q_3)$
$const_2(q_2) = f_1 \wedge f_2 \wedge (\neg(f_3) \vee const_1(q_3)) = (f_1 \vee \neg f_3) \wedge (f_2 \vee \neg f_3) \wedge (f_1 \wedge f_2)$
$= const_3(q_2)$
$const_3(q_1) = f_1 \wedge \big(const_2(q_2)\big) \wedge \neg(f_1 \wedge \neg f_2) = f_1 \wedge (f_1 \vee \neg f_3) \wedge (f_2 \vee \neg f_3) \wedge$
$(f_1 \wedge f_2) \wedge (\neg f_1 \vee f_2) = const_4(q_1)$
$const_4(q_0) = \top \wedge (\neg f_1 \vee (const_3(q_1))) = (\neg f_1 \vee f_2) = const_5(q_0)$

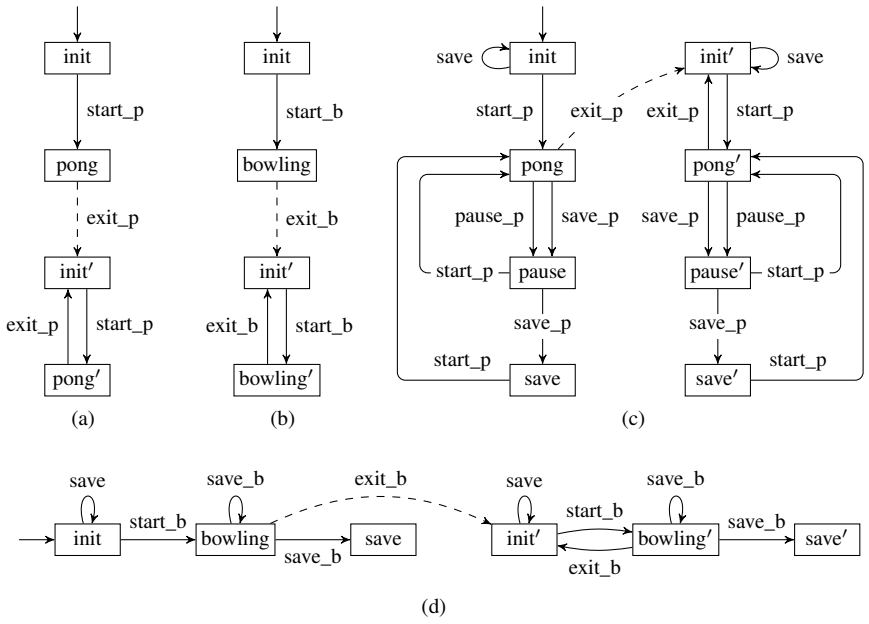


Figure 7: FTS of Figure 3 Translated into a Set of MTSs

Considering the fixed points in the above computations, it holds

$$\Lambda_{mts} = \{\neg f_1 \wedge f_2 \wedge f_3, \neg f_1 \wedge \neg f_2 \wedge f_3, \neg f_1 \wedge f_2 \wedge \neg f_3,$$

$$\neg f_1 \wedge \neg f_2 \wedge \neg f_3, f_1 \wedge f_2 \wedge f_3, f_1 \wedge f_2 \wedge \neg f_3\}.$$

By performing similar computations for mts', we can conclude from Definition 8 that $\{mts, mts'\}$ is in the context of the FTS in Figure 4d. As another example, consider Figure 7 to represent the MTSs in the context of the FTS in Figure 3. Note that we, again, have to perform loop unrolling here in order to obtain the correct MTS encoding as described before.

We next prove that, given a set of MTSs in the context of a given FTS according to Definition 9, the union of the sets of products implemented by the MTSs in this set is equal (up to bisimulation) to the set of products implemented by the FTS. In both cases, the product implementations are represented as LTSs. To this end, we first define the (set of) configuration vector(s) corresponding to an LTS implementing an MTS that belongs to a set of MTSs in the context of the given FTS. This definition is then used in the proof of Theorem 1. Intuitively, the construction of this set follows the same idea as the one given in Definition 7.

Definition 10. *Given an MTS* $mts = (Q, A, \rightarrow_\Diamond, \rightarrow_\Box, q_{init})$ *from a set of MTSs in the context of a given FTS* $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$. *For each LTS* $lts = (S, A, \rightarrow, s_{init})$ *being a valid implementation of* mts, *the (set of) corresponding configuration vector(s)*

is defined as follows. Considering lts, there exists a class of relations, denoted by $\mathcal{R} \subseteq 2^{S \times Q}$, where each $\mathcal{R}^i \in \mathcal{R}$ is a refinement relation (cf. Definition 3) that relates states of lts to states of mts. We first define the following auxiliary sets:

$$impMust^i = \{(p,e) \xrightarrow{a}_{\Box} (p',e') \mid \exists_{s,s' \in S, a \in A} \cdot\ (s,a,s') \in \rightarrow\ \wedge\ s\mathcal{R}^i(p,e) \wedge s'\mathcal{R}^i(p',e')\}$$

$$impMay^i = \{(p,e) \xrightarrow{a}_{\Diamond} (p',e') \mid \exists_{s,s' \in S, a \in A} \cdot\ (s,a,s') \in \rightarrow\ \wedge\ s\mathcal{R}^i(p,e) \wedge s'\mathcal{R}^i(p',e')\}$$

$$impExc^i = \{(p,f,a,p') \in \rightarrow\ \mid \big(\exists((p,e),a,(p',e \wedge f)) \in \rightarrow_{\Diamond} \cdot (\nexists s' \cdot\ (s,a,s') \in \rightarrow \wedge\ s'\mathcal{R}^i(p',e')\big)\ \vee \big(\exists (p,e) \in Q \cdot\ Sat(e \wedge f)\ \wedge\ ((p,e),a,(p',e \wedge f)) \notin \rightarrow_{\Diamond}\ \big)\}$$

Given a refinement relation $\mathcal{R}^i \in \mathcal{R}$, the (set of) configuration vector(s) corresponding to lts is defined as:

$$\Lambda_{lts} = \{\lambda \in \Lambda_{mts} \mid \lambda\ \Longrightarrow\ conf^i(lts)\},$$

where $conf^i(lts)$ is defined as:

$$conf^i(lts) = \bigwedge_{(p,e) \xrightarrow{a}_{\Box} (p',e') \in impMust^i} e'\ \wedge \bigwedge_{(p,e) \xrightarrow{a}_{\Diamond} (p',e') \in impMay^i} e'\ \wedge \bigwedge_{(p,f,a,p') \in \rightarrow \in impExc^i} \neg f$$

Based on this definition, we are now able to prove the correctness of our encoding from FTSs into MTSs.

In particular, we can reduce this problem to a mutual comparison of the sets of LTSs corresponding to product implementations derivable from both representations.

Theorem 1. *Given an FTS fts, the set of LTSs implementing fts is equal to the union of sets of LTSs implementing each sets of MTSs being the context of fts, i.e.*

$$\forall_{\mathcal{M} \in context(fts)}\ [\![fts]\!] = \bigcup_{mts \in \mathcal{M}} [\![mts]\!].$$

Proof. We divide the proof into two following obligations, one for each direction. First, we prove that $[\![fts]\!] \subseteq \bigcup_{mts \in \mathcal{M}} [\![mts]\!]$ holds. Given $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$, the set of MTSs $\mathcal{M} \in context(fts)$ and an LTS $lts = (S, A, \rightarrow, s_{init})$, s.t. $lts \in [\![fts]\!]$. We prove $[\![fts]\!] \subseteq \bigcup_{mts \in \mathcal{M}} [\![mts]\!]$ by showing that

$$\exists_{mts \in \mathcal{M}}\ lts \in [\![mts]\!]. \tag{1}$$

To prove $lts \in [\![mts]\!]$, and assuming that $mts = (Q, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, q_{init})$, based on Definition 3, it suffices to show that a refinement relation such as $\mathcal{R}_{mts} \subseteq Q \times S$ exists such that

1. $(q_{init}, s_{init}) \in \mathcal{R}_{mts}$
2. $\forall_{q\in Q,s,s'\in S,a\in A}\ \big(q\mathcal{R}_{mts}s\ \wedge\ s \xrightarrow{a} s'\big)\implies\ \exists_{q'\in Q}\ q \xrightarrow{a}_{\Diamond} q' \wedge q'\mathcal{R}_{mts}s'.$
3. $\forall_{q,q'\in Q,s\in S,a\in A}\ \big(q\mathcal{R}_{mts}s\ \wedge\ q \xrightarrow{a}_{\Box} q'\big)\implies\ \exists_{s'\in S}\ s \xrightarrow{a} s' \wedge q'\mathcal{R}_{mts}s'.$

We choose $mts = (Q, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, q_{init})$ from $\mathcal{M}$ such that $\lambda \in \Lambda_{mts}$, where λ is a product configuration for which there exists $\mathcal{R}_{\lambda} \subseteq \mathbb{P} \times S$ that satisfies:

i $(p_{init}, s_{init}) \in \mathcal{R}_{\lambda}$

ii $\forall_{p,p'\in\mathbb{P},a\in A,s\in S,f\in\mathbb{B}(F)}\ \big(p\ \mathcal{R}_{\lambda}\ s\ \wedge\ p \xrightarrow{f/a} p'\ \wedge\ \lambda \models f\big)\ \Rightarrow\ \exists_{s'\in S}\cdot\ s \xrightarrow{a} s'\ \wedge\ p'\ \mathcal{R}_{\lambda}\ s'.$

iii $\forall_{p\in\mathbb{P},a\in A,s,s'\in S}\ \big(p\ \mathcal{R}_{\lambda}\ s\ \wedge\ s \xrightarrow{a} s'\big)\ \Rightarrow\ \exists_{p'\in\mathbb{P},f\in\mathbb{B}(F)}\cdot\ p \xrightarrow{f/a} p'\ \wedge\ \lambda \models f \wedge\ p'\ \mathcal{R}_{\lambda}\ s'.$

Given $lts \in [\![fts]\!]$ based on Definition 6 as defined above, a $\mathcal{R}_{\lambda}$ with properties (i), (ii), and (iii) exists for some $\lambda \in \Lambda$. Furthermore, based on the first condition in Definition 9, there exists $mts \in \mathcal{M}$ such that $\lambda \in \Lambda_{mts}$.

Given $\mathcal{R}_{\lambda}$ with the above properties, we define a relation $\mathcal{R}_{mts}$ as follows:

$$\forall_{(p,e)\in Q,s\in S}\ (p, e)\ \mathcal{R}_{mts}\ s \Leftrightarrow p\ \mathcal{R}_{\lambda}\ s\ \wedge\ \lambda \models e \tag{2}$$

Next, we prove that $\mathcal{R}_{mts}$ satisfies property (1). As $\lambda \models \bigvee_{\lambda\in\Lambda} \lambda$ and $p_{init}\mathcal{R}_{\lambda}s_{init}$, based on the definition of $\mathcal{R}_{mts}$, it holds that $q_{init}\mathcal{R}_{mts}s_{init}$. Hence, property (1) holds.

We further prove that R_{mts} satisfies property (2). Consider an arbitrary pair of states $((p, e), s) \in \mathcal{R}_{mts}$. Based on property (iii) it holds that

$$\big(p\ \mathcal{R}_{\lambda}\ s\ \wedge\ s \xrightarrow{a} s'\big)\ \Rightarrow\ \exists_{p'\in\mathbb{P},f\in\mathbb{B}(F)}\cdot\ p \xrightarrow{f/a} p'\ \wedge\ \lambda \models f \wedge\ p'\ \mathcal{R}_{\lambda}\ s'.$$

From $\lambda \in \Lambda_{mts}$ and from Definition 7 we conclude that each transition $p \xrightarrow{f/a} p'$ on the right hand side of the above statement with $\lambda \models f$ is translated into a may transition mts emanating state (p, e) (cf. Lemma 2). Hence, property (2) holds.

Next, we prove that R_{mts} satisfies property (3). Consider an arbitrary pair of states $((p, e), s) \in \mathcal{R}_{mts}$ and a must-transition $(p, e) \xrightarrow{a}_{\Box} (p, e')$. Based on Definition 8, this transition correspond to a transition $p \xrightarrow{f/a} p'$ in the *fts* such that $e' = e \wedge f$. Hence, from the respective definition of *const*, it follows that $\lambda \models e$ and $\lambda \models f$. In addition, based on the condition given in property (ii) and the definition of $\mathcal{R}_{mts}$ in Equation (2), it holds that

$$\exists_{s'\in Q}\ s \xrightarrow{a} s'\ \wedge\ (p', e')\ \mathcal{R}_{mts}\ s'.$$

Thus, property (3) holds. Second, we prove $\bigcup_{mts\in\mathcal{M}}[\![mts]\!] \subseteq [\![fts]\!]$. Given $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$ and the set of MTSs $\mathcal{M} \in context(fts)$. Consider $mts{=}(Q, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, q_{init})$ such that $mts \in \mathcal{M}$, and an LTS $lts = (S, A, \rightarrow, s_{init})$ such that $lts \in [\![mts]\!]$. In order to prove $\bigcup_{mts\in\mathcal{M}}[\![mts]\!] \subseteq [\![fts]\!]$ it suffices to show that

$$lts \in [\![fts]\!] \tag{3}$$

holds. Given the above definitions for *fts* and *lts* and the definition of product derivation for FTSs (cf. Definition 6), it is sufficient to show that for a product configuration $\lambda \in \Lambda$ a relation $\mathcal{R}_\lambda \subseteq \mathbb{P} \times S$ exists such that the following holds:

1. $(p_{init}, s_{init}) \in \mathcal{R}_\lambda$.
2. $\forall_{p,p' \in \mathbb{P}, a \in A, s \in S, \varphi \in \mathbb{B}(F)} \left(p \; \mathcal{R}_\lambda \; s \; \wedge \; p \xrightarrow{\varphi/a} p' \; \wedge \; \lambda \models \varphi \right) \; \Rightarrow \; \exists_{s' \in S} \cdot \; s \xrightarrow{a} s' \; \wedge \; p' \; \mathcal{R}_\lambda \; s'$.
3. $\forall_{p \in \mathbb{P}, a \in A, s, s' \in S} \left(p \; \mathcal{R}_\lambda \; s \; \wedge \; s \xrightarrow{a} s' \right) \; \Rightarrow \; \exists_{p' \in \mathbb{P}, \varphi \in \mathbb{B}(f)} \cdot \; p \xrightarrow{\varphi/a} p' \; \wedge \; \lambda \models \varphi \wedge \; p' \; \mathcal{R}_\lambda \; s'$.

Given $lts \in [\![mts]\!]$ and, based on Definition 10, there exists a set of refinement relations such as $\mathcal{R} = \bigcup_{1 \leq i \leq n} \mathcal{R}^i_{mts}$, where for each relation $\mathcal{R}^i_{mts}$, the following properties hold (see Definition 3).

i $(q_{init}, s_{init}) \in \mathcal{R}^i_{mts}$

ii $\forall_{q,q' \in Q, s \in S, a \in A} \left(q \mathcal{R}^i_{mts} s \; \wedge \; q \xrightarrow{a}_{\Box} q' \right) \implies \exists_{s' \in S} \; s \xrightarrow{a} s' \wedge q' \mathcal{R}^i_{mts} s'$.

iii $\forall_{q \in Q, s, s' \in S, a \in A} \left(q \mathcal{R}^i_{mts} s \; \wedge \; s \xrightarrow{a} s' \right) \implies \exists_{q' \in Q} \; q \xrightarrow{a}_{\Diamond} q' \wedge q' \mathcal{R}^i_{mts} s'$.

For $\lambda \in \Lambda_{lts}$ and relation $\mathcal{R}^i_{mts} \in \mathcal{R}$, we define $\mathcal{R}^i_\lambda$ as follows.

$$s \; \mathcal{R}^i_\lambda \; p \leftrightarrow \exists (p, e) \in Q \cdot s \; \mathcal{R}^i_{mts} \; (p, e) \; \wedge \; (\lambda \implies conf^i(lts)) \; \wedge \; \lambda \models e \qquad (4)$$

First, we prove that $\mathcal{R}^i_\lambda$ satisfies property (1). Based on Definition 8, we have $q_{init} = (p_{init}, \bigvee_{\lambda \in \Lambda} \lambda)$ and thus $\lambda \models \bigvee_{\lambda \in \Lambda}$. Given property (i) and the definition of $\mathcal{R}^i_\lambda$ given in Equation (4), it holds that $\mathcal{R}^i_\lambda$ satisfies property (1).

Next, we prove that $\mathcal{R}^i_\lambda$ satisfies property (2). Consider a pair of states $(s, p) \in \mathcal{R}^i_\lambda$ and a transition $p \xrightarrow{f/a} p'$, where $\lambda \models f$. We can assume the following three cases.

- The transition is included as must-transition in mts. Based on property (ii), for all such transitions there exists $s \xrightarrow{a} s'$ such that $s' \mathcal{R}^i_{mts} (p', e \wedge f)$. According to the definition of $\mathcal{R}^i_\lambda$ given in Equation (4), we have that $\lambda \models e$ and as $\lambda \models f$ holds, it further holds that $\lambda \models e \wedge f$. Hence, considering $\lambda \in \Lambda_{lts}$, due to Equation (4) it holds that $s' \mathcal{R}^i_\lambda p'$.
- The transition is included as a may-transition in mts. Hence, we have $(p, e) \xrightarrow{a}_{\Diamond} (p', e \wedge f)$. Given $lts \in [\![mts]\!]$, due to property (iii) it holds that $s \xrightarrow{a} s'$ such that $s' \mathcal{R}^i_{mts} (p', e \wedge f)$. Otherwise, according to Definition 10, $\neg f$ would be part of the conjunction included in the construction of $conf^i(lts)$, which results in $\lambda \not\models f$. According to the definition of $\mathcal{R}^i_\lambda$ given in Equation (4), it holds that $\lambda \models e$ and, based on the assumption $\lambda \models f$, it follows that $\lambda \models e \wedge f$. Hence, considering $\lambda \in \Lambda_{lts}$, due to Equation (4) it holds that $s' R^i_\lambda p'$.
- The transition excluded from mts. This case is not valid according to Lemma 2.

Based on the three above cases, we conclude that R^i_λ satisfies property (2). Finally, we prove that $\mathcal{R}^i_\lambda$ satisfies property (3). We consider a pair of states $(s,p) \in \mathcal{R}^i_\lambda$ and a transition $s \xrightarrow{a} s'$. Based on property (iii), there exists a transition $(p,e) \xrightarrow{a}_\Diamond (p',e')$ in the mts such that $s'\ \mathcal{R}^i_{mts}(p',e')$ holds. Based on the Definition 8, each transition $(p,e) \xrightarrow{a}_\Diamond (p',e')$ results from encoding a transition $p \xrightarrow{f/a} p'$. Based on Equation (4), it holds that $\lambda \implies conf^i(lts)$. Given the construction of $conf^i(lts)$ in Definition 10, it holds that $\lambda \models f$. Considering that $\lambda \models e$ holds, it can be concluded from Equation (4) that $s'\mathcal{R}^i_\lambda p'$ holds. Hence, $\mathcal{R}^i_\lambda$ also satisfies property (3). □

5. Generating Minimal MTS Encodings of FTSs

The MTS encoding of FTSs as defined in the previous section always permits, as a trivial solution, to simply interpret a given FTS as a set of LTSs (i.e., MTSs with $\rightarrow_\Diamond = \rightarrow_\Box$) corresponding to the set of valid implementations of the FTS. This solution may be considered as the *maximal* encoding.

In this section, we provide a constructive algorithm for computing the declarative definition of context (Definition 9) and prove its correctness. Furthermore, we show that using this algorithm, we not only generate valid, but also *minimal* (i.e., most succinct [12]) MTS encodings of a given FTS.

5.1. Generation of MTS Encodings

An operational characterization of generating an MTS encoding from a given FTS as defined in a declarative manner in the previous section is given in Algorithm 1. The algorithm receives as input an FTS *fts* and returns as output a set $\mathcal{M}$ of MTSs being an MTS encoding of *fts*. We describe the two procedures MAIN (lines 1–7) and NEWMTS (lines 8–45) in more detail in the following.

Procedure MAIN. The procedure MAIN repeatedly calls procedure NEWMTS for generating further MTSs to be added to the result set $\mathcal{M}$ until every valid implementation of *fts* is finally covered by some LTS variant of at least one MTS in the set $\mathcal{M}$. First, result set $\mathcal{M}$ is initialized as empty set (line 2). Next, a presence condition $m \in \mathbb{B}(F)$ is introduced (line 3). This so-called *blocking clause* is used throughout the algorithm to represent the set of configurations which are *not* yet covered by some MTS within the current result set $\mathcal{M}$ (i.e., initially all configurations $\lambda \in \Lambda$ of *fts*, cf. line 3). The main loop then adds further MTS into result set $\mathcal{M}$ until the set of not-yet-covered configurations is empty (i.e., the blocking clause becomes unsatisfiable, cf. line 4). To this end, procedure NEWMTS is invoked with the current blocking clause m and returns a further MTS *mts* to be added to $\mathcal{M}$ (line 6) together with a feature expression *blockingClause* defining the set of additional configurations covered by the new MTS (line 5). Hence, the blocking clause m is updated by conjunction of the negated *blockingClause* expression before starting the next iteration.

Algorithm 1 MTS Generation

Input: $\mathit{fts} := (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{\mathit{init}})$
Output: $\mathcal{M} := \{(Q^i, A^i, \rightarrow^i_\Diamond, \rightarrow^i_\Box, q^i_{\mathit{init}})\}$

1: **procedure** MAIN
2: $\mathcal{M} := \emptyset$
3: $m := \bigvee_{\lambda \in \Lambda} \lambda$
4: **while** m **do**
5: $(\mathit{mts}, \mathit{blockingClause}) :=$ NEWMTS(m)
6: $\mathcal{M} = \mathcal{M} \cup \{\mathit{mts}\}$
7: $m := m \wedge \neg \mathit{blockingClause}$

8: **procedure** NEWMTS(featureExpression m)
9: $\mathit{fts}_m := (\mathbb{P}, A, F, \rightarrow_m, \Lambda_m, p^m_{\mathit{init}})$, where $\rightarrow_m := \emptyset$, $\Lambda_m := \{\lambda \in \Lambda \mid \lambda \vdash m\}$ and $p^m_{\mathit{init}} := p_{\mathit{init}}$
10: $\mathit{mts}_m := (\mathbb{P} \times \mathbb{B}(F), A, \emptyset, \emptyset, (p^m_{\mathit{init}}, m))$
11: $\rightarrow_x := \emptyset$, $DS := \emptyset$, $\mathcal{T} := \emptyset, \mathcal{U} := \emptyset$
12: $WS := \{(p_{\mathit{init}}, m)\}$
13: **while** $WS \neq \emptyset$ **do**
14: $q := (p, e) \in WS$, $WS := WS \setminus \{q\}$
15: $DS := DS \cup \{q\}$
16: **for each** $(p, a, f, p') \in \rightarrow$ **do** // iterate over all outgoing transitions of FTS state p
17: $\rightarrow_m := \rightarrow_m \cup \{(p, f, a, p')\}$
18: $\rightarrow_\Diamond := \rightarrow_\Diamond \cup \{((p, e), a, (p', e \wedge f))\}$
19: **if** $\neg(\bigwedge_{\{f \mid \exists((p,e),a,(p',e\wedge f)) \in \rightarrow_\Diamond\}} f) \wedge m$ // check if transition is incompatible with others
20: $\rightarrow_x := \rightarrow_x \cup \{((p, e), a, (p', e \wedge f))\}$
21: $\rightarrow_\Diamond := \rightarrow_\Diamond \setminus \{((p, e), a, (p', e \wedge f))\}$
22: **else**
23: $\mathcal{T} :=$ UPDATETOPOREL$(\mathcal{T}, \mathit{mts}_m)$
24: $\mathcal{U} :=$ UPDATEUNROLLEDOPTIONALLOOPS$(\mathcal{U}, \mathit{mts}_m)$
25: **if** $\exists((p, e'), a, (p', e' \wedge f)) \in \rightarrow_\Box: (((p, e'), a, (p', e' \wedge f)), ((p, e), a, (p', e \wedge f))) \in \mathcal{T} \wedge \neg(m \Rightarrow f)$ // check for unnecessary unrolling of mandatory transitions
26: **return** NEWMTS$(m \wedge f)$
27: // check if presence condition is implied by other presence conditions and/or initial condition:
28: **if** $(\exists((p'', e''), a'', (p''', e'' \wedge f'')) \in \rightarrow_\Diamond: f'' \wedge \mathit{const}_i(p^m_{\mathit{init}}, m) \Rightarrow f)$
29: $\rightarrow_\Box := \rightarrow_\Box \cup \{((p, e), a, (p', e \wedge f))\}$
30: **if** $((p, e), a, (p', e \wedge f)) \in \mathcal{U}$ // check for unnecessary unrolling
31: **return** NEWMTS$(m \wedge f)$
32: **for each** $s \in \mathcal{P}(\rightarrow_\Diamond \setminus \rightarrow_\Box)$ **do** // check combinations of optional transitions
33: **if** $\neg((\bigwedge_{\{f \mid \exists((p,e),a,(p',e\wedge f)) \in s\}} \neg f) \wedge (\bigwedge_{\{f \mid \exists((p,e),a,(p',e\wedge f)) \in (\rightarrow_\Diamond \setminus \rightarrow_\Box) \setminus s\}} f))$
34: $\rightarrow_\Box := \rightarrow_\Box \cup \{((p, e), a, (p', e \wedge f)) := \text{PICKELEMENT}(s)\}$
35: **if** $((p, e), a, (p', e \wedge f)) \in \mathcal{U}$ // check for unnecessary unrolling
36: **return** NEWMTS$(m \wedge f)$
37: **break**
38: // check if transition is dependent on other transitions:
39: **while** $(\exists((p, e), a, (p', e \wedge f)) \in \rightarrow_\Diamond \setminus \rightarrow_\Box:$ $((\exists((p'', e''), a'', (p''', e'' \wedge f'')) \in \rightarrow_\Diamond \setminus \{t \mid t \in \rightarrow_\Diamond \wedge (t, ((p, e), a, (p', e \wedge f))) \in \mathcal{T}\} :$ $(f'' \wedge \mathit{const}_i(p^m_{\mathit{init}}, m)) \Rightarrow f)))$ **do**
40: $\rightarrow_\Box := \rightarrow_\Box \cup \{((p, e), a, (p', e \wedge f))\}$
41: **if** $((p, e), a, (p', e \wedge f)) \in \mathcal{U}$ // check for unnecessary unrolling
42: **return** NEWMTS$(m \wedge f)$
43: **if** $\nexists(p', e'') \in DS$
44: $WS := WS \cup \{(p', e \wedge f)\}$
45: **return** $(\mathit{mts}_m, \mathit{const}_i(p_{\mathit{init}}, m))$

Procedure NEWMTS. The procedure NEWMTS constructs the next MTS mts_m from FTS *fts* with respect to the current blocking clause m by starting with an empty fts_m and by incrementally traversing (and potentially adding) all transitions of *fts* into fts_m being reachable from the initial state. Each traversed transition either becomes a may-transition, a must-transition or an excluded transition in mts_m, depending on the presence conditions of the previously added transitions. To this end, a call to the helper-function $const_i$ returns the MTS constraint denoting the maximal fixed point (cf. Definition 7) with respect to the current (intermediate) models mts_m and fts_m.

First, fts_m is initialized without any transitions and the set of configurations being restricted by the blocking clause m (line 9). Similarly, mts_m is also initialized with no transitions (line 10). Furthermore, additional temporary data structures are initialized, namely a set $\rightarrow_x$ to store those transitions from *fts* being excluded from mts_m and a set *DS* (*done-set*) to store those states of mts_m already visited during the traversal (line 11). Moreover, we utilize the relation $\mathcal{T} \subseteq \rightarrow_\Diamond \times \rightarrow_\Diamond$ containing those pairs of transitions of an MTS being in a *topological order* (i.e., $(t, t') \in \mathcal{T}$ either iff transition t always precedes transition t' on every path leading from the initial state to the first occurrence of t', or $t = t'$ holds). The worst-case complexity of computing $\mathcal{T}$ is quadratic in the number of transitions. In addition, we further initialize a set $\mathcal{U} \subseteq \rightarrow_\Diamond$ for memorizing those transitions from the FTS being added as unrolled (optional) transition into the MTS under construction as described in Section 4. This set is used to check whether such previously performed unrollings may become obsolete in subsequent steps of the MTS construction due to dependencies among presence conditions of FTS transitions involved (see below for details).

In addition, the set *WS* (*working-set*) is used to store those (still-to-be-processed) states of mts_m that are directly reachable via previously added states in mts_m, either by optional or mandatory transitions. This set initially contains the initial state of *fts*, being restricted by m (line 12). The main iteration (line 13) then proceeds as long as the working-set *WS* contains further states, by picking-and-removing an arbitrary next state $q = (p, e)$ from *WS* (line 14) and by adding component q (i.e., the respective state in *fts*) into the done-set *DS*.

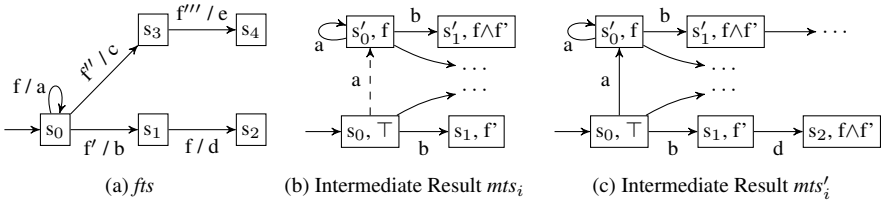
Next, we iterate over the set of all outgoing transitions of state q in *fts* (line 16) and add them to fts_m (line 17). We first try to add those transitions as a may-transitions into mts_m (line 18). Here, $(p', e \wedge f)$ denotes an MTS state in which the presence condition $e \wedge f$ holds (i.e., if there already exists a state (p', e') in the MTS with e' being equivalent to $e \wedge f$, the target state of the newly added transition is that existing state). In the next step, we check whether adding the currently considered transition of the FTS would lead to an inconsistent MTS model (line 19). This is done by checking compatibility of the conjunction of all presence conditions of transitions from *fts* already added as may-transitions into mts_m in previous steps including the current one. In case of non-satisfiability, the transition is instead added to the exclude-set (line 20) and removed from the set of may-transitions of mts_m (but it remains in the set of transitions of fts_m in order to mark it as already processed and explicitly excluded). In case the condition in line 19 is not satisfied (i.e., the new transition can be added to mts_m), we incrementally update relation $\mathcal{T}$ (line 23) and set $\mathcal{U}$ (line 24) of mts_m by taking the newly added transition into account.

Furthermore, we have to prevent unnecessary loop unrollings which may occur in

two different possible ways throughout the construction steps performed by the algorithm up to this point.

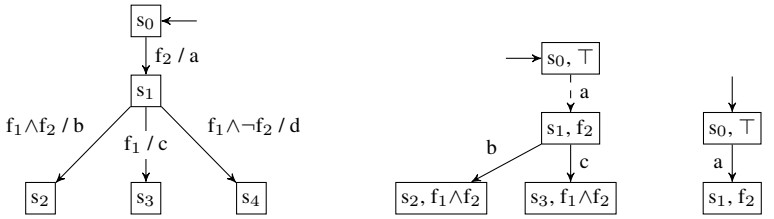
The first case occurs if an FTS transition added as mandatory transition into the MTS in a previous step (due to dependencies of its presence conditions to those other transitions) is additionally added as unrolled transition in a subsequent step. For instance, this unrolling may happen if the previously added mandatory transition is followed by an optional transition being located within the same loop (including that transition itself). As adding the optional transition may result in an updated path condition not being equivalent to the path condition holding at the beginning of the loop, the loop will be unrolled in the MTS. However, to handle those cases, we have to distinguish between transitions being (correctly) mandatory solely due to their presence condition and transitions (incorrectly) becoming mandatory due to dependencies between their presence conditions to those of other mandatory transitions thus potentially resulting in unnecessary unrollings as described above. In order to avoid the latter case, we check for each newly added transition in the MTS if the corresponding FTS transition has already been inserted into the MTS in a previous step (as a mandatory transition) thus encountering a case of loop unrolling (cf. line 25). If this is the case, we restart the current call of procedure NEWMTS (cf. line 26) with an adapted initial condition such that the (falsely) unrolled transition immediately becomes mandatory. To avoid infinite recursion in case of correctly unrolled mandatory transitions, we further have to check if its presence condition is already implied by the initial condition before restarting (cf. second part of line 25).

The second case occurs if an FTS transition added as unrolled optional transition into the MTS in a previous step later becomes mandatory due to dependencies of its presence conditions to those of other transitions added in subsequent steps. Hence, whenever a transition becomes mandatory (lines 29, 34 and 40), we have to check whether this transition is part of an unrolled loop in the MTS (lines 30, 35 and 41). If this is the case, we also restart NEWMTS (lines 31, 36 and 42), again, by additionally conjuncting the presence condition of the respective transition to the initial condition (thus making the transition to an a-priori mandatory transition). Hence, the loop now only consists of mandatory transitions and is therefore not unrolled anymore as the path condition holding after the loop is equivalent to the path condition already holding at the beginning of the loop. Figure 8 provides an example for the necessity of this restart (where all features are assumed optional). Here, state s_0 of the FTS depicted in Figure 8a has a self-loop transition which will be unrolled as feature f is optional. As a result, all following transitions will be duplicated, too. For instance, mts_i (cf. Figure 8b) illustrates an intermediate result where the transition labeled with a is unrolled such that the transition labeled with b is duplicated and therefore becomes mandatory. Note that the first transition labeled a is optional such that whenever a is included in an MTS variant, it may be performed arbitrarily often afterwards (as induced by the FTS). When proceeding the constructions in Algorithm 1, we will reach the transition labeled with d at some point. As this transition has the same presence condition as the transition labeled with a, both transitions will then become mandatory as there are only variants permitted having either both a and d included or none of them. Hence, the previously unrolled optional transition labeled with a becomes mandatory and we restart NEWMTS with the refined feature condition $m = \top \wedge f$. As a result, the loop of



(a) *fts* (b) Intermediate Result mts_i (c) Intermediate Result mts'_i

Figure 8: Example for the Necessity of Restarting NEWMTS



(a) *fts* (b) mts_1 (c) mts_2

Figure 9: Example for Algorithm 1

the initial state is not unrolled anymore (as $\top \wedge f$ is obviously equivalent to $\top \wedge f \wedge f$) and therefore no transitions will be duplicated. As a consequence, the transition labeled with b (amongst others) also remains optional. In this way, we only restart NEWMTS if we encounter cases of (unnecessarily) unrolled transitions (e.g., the transition with label a), but not in case of (necessarily) duplicated transitions due to (necessary) unrollings.

Furthermore, we check whether the presence condition of the newly added transition is implied by either the presence condition of some other FTS transition already added to mts_m or by the current MTS constraint of mts_m (line 28). If one of these two cases holds, then the newly added transition has to become mandatory (line 29). Figure 9 provides an example for this issue (note that Example 1 on page 24 provides a full description of applying Algorithm 1 to this example FTS). While generating the MTS in Figure 9b, we have an intermediate step where the transitions labeled with actions a and c are optional and the transition labeled with action b is mandatory. As a consequence, the transition labeled with action c has to become mandatory, too, as the presence condition of the respective FTS transition labeled with action c (cf. Figure 9a) is implied by the presence condition of the FTS transition labeled with action b. Hence, every variant containing the transition labeled with action b must also contain the transition labeled with action c.

In addition, we have to check whether an optional transition has to become mandatory if this is implied by a particular *combination* of other optional transitions (lines 32 to 37). For this, we consider each subset of optional transitions (line 32, where $\mathcal{P}$ denotes power set) in ascending order, starting with the smallest sets. First, we conjugate the negated presence conditions of all transitions being in s, and then conjugate the

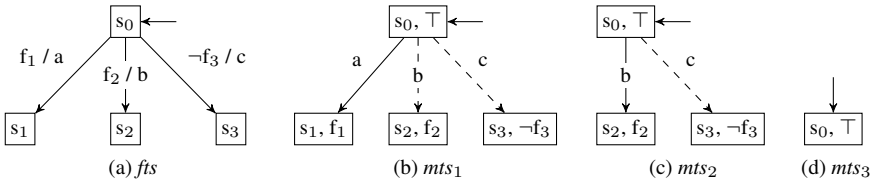


Figure 10: Example for Lines 32 to 34 of Algorithm 1 with Feature Model $m = f_1 \vee f_2 \vee f_3$

presence conditions of all optional transitions not being in s (line 33) and check the resulting formula for satisfiability. If this check fails, then the corresponding combination of optional transitions is not permitted to *not* include those being in s in a valid variant while including those being in s. We then pick one element from s to become mandatory (line 34) and are then able to immediately terminate the check due to the ascending traversal (line 37).

Figure 10 (with feature model $m = f_1 \vee f_2 \vee f_3$) provides an example for the checks in lines 32 to 37. Here, a variant only containing a transition labeled with action c is the only variant not permitted by the FTS. Without the check in lines 32 to 37, Algorithm 1 would produce mts_1 with all transitions being optional. However, by adding this check, subset s containing the transitions labeled with actions a and b will yield the unsatisfiable formula $\neg f_1 \wedge \neg f_2 \wedge \neg f_3 \wedge (f_1 \vee f_2 \vee f_3)$. As a result, we will pick either the transitions labeled with a or with b and make it mandatory thus leading to a correct solution. The same holds for mts_2 which is generated next. Without lines 32 to 37, both transitions would be optional, thus again allowing an (invalid) variant only containing a transition labeled with action c.

In the next step, we have to check whether the update of mts_m (i.e., either adding a transition from *fts* as optional or mandatory transition or excluding it from mts_m) potentially causes existing optional transitions in mts_m to become mandatory. For this, we have to check for each optional transition if the presence condition of the corresponding transition from fts_m is implied by (combinations of) presence conditions of other transitions added to mts_m (which is, again, done by invoking $const_i$ on the current model in line 39). In this case, the set of must-transitions of mts_m is updated, accordingly (line 40). However, for obtaining the minimal solution, this only holds for transitions not topologically preceding the transition under consideration. In those cases, reachability of the current transition already depends on the presence/absence of the topologically preceding transition. Figure 9 gives an example for this exception. Here, the transition labeled a of the MTS in Figure 9b will not become mandatory although the respective presence condition of the FTS (cf. Figure 9a) is implied by the presence condition of b.

Finally, we have to check whether the target state of the currently processed transition is already subsumed by some state in the done-set *DS* (line 43). Otherwise, we add the state into the working-set *WS* (line 44).

If the working-set contains no further states, mts_m together with the corresponding MTS constraint is finally returned to procedure MAIN.

Example 1. *Figure 9 provides an example for a complete application of Algorithm 1.*

Here, Figure 9a depicts the fts and Figures 9b and 9c show the resulting set of MTSs. After initialization, it holds that $m = \top$ as both f_1 and f_2 are optional features being independent from each other, thus invoking NEWMTS$(\top)$. *This procedure starts with $q = (s_0, \top)$ as initial state in the working set WS. State s_0 has one outgoing transition in fts labeled a, which is therefore added to mts_1 as a may-transition as well as to fts_m (lines 16 to 18), whereas the set of excluded transitions of mts_1 remains empty (lines 19 to 21). As this newly added may-transition does not (yet) depend on any other transition in mts_1 or on blocking clause m, it does not become mandatory (lines 39 and 40). Next, the target state (s_1, f_2) of the transition is added to the working set WS (line 43 and 44). As state $q = (s_0, \top)$ does not have further outgoing transitions (line 16) and WS is not empty (line 13), the next iteration of the while-loop starts, for instance, by picking the transition labeled with b (line 16). Here, lines 17 to 21 yield similar results as before. Additionally, the while-loop in line 39 does not add a new must-transition although the presence condition f_2 of the previously added transition labeled a is implied by the presence condition $f_1 \wedge f_2$ of the newly added transition labeled b. This is due to the transitions of a and b being in the topological relation, i.e., every path leading to b also visits a. Therefore, a may remain optional. When the transition labeled c is added to mts_1, it becomes mandatory as the presence condition of b implies the presence condition of c (line 28 to 29). Furthermore, b becomes mandatory as there is no variant with b but without a (lines 32 to 37). A variant with a and c (as c is mandatory) has features f_1 and f_2 selected, and hence, c must be included in this variant as well. As a result, the new fixed point is now given as const$(s_0, \top) = \neg f_2 \vee (f_1 \wedge f_2)$. In contrast, the transition labeled d has to be excluded from mts_1 as its presence condition is not compatible with those of the transitions labeled a and b (line 19). The new fixed point is therefore given as*

$$\mathit{const}(s_0, \top) = (\neg f_2 \vee (f_1 \wedge f_2)) \wedge \neg(f_1 \wedge \neg f_2)$$

which is equivalent to $\neg f_2 \vee (f_1 \wedge f_2)$. This leads to termination of NEWMTS *with mts_1 and the respective fixed point being returned (line 45). The next invocation* NEWMTS$(\top \wedge \neg(\neg f_2 \vee (f_1 \wedge f_2)))$ *(being equivalent to $\neg f_1 \wedge f_2$) then returns mts_2 as shown in Figure 9c thus causing the updated blocking clause m to become unsatisfiable and procedure* MAIN *to terminate with $\mathcal{M} = \{mts_1, mts_2\}$.*

We now prove correctness of Algorithm 1 with respect to the definition of MTS encoding of FTS (cf. Definition 8).

Theorem 2. *Let $\mathcal{M}$ be the MTS encoding of an FTS fts as generated by Algorithm 1. Then it holds that $\mathcal{M} \in$ context(fts).*

Proof. We prove Theorem 2 by showing that (1) $\forall mts \in \mathcal{M} : (\forall lts \preceq mts : fts \triangleright lts)$ and (2) $\forall lts \cdot\ fts \triangleright lts : (\exists mts \in \mathcal{M} : lts \preceq mts)$.

1. Proof by induction. Initially, mts_m contains no transitions. In each iteration of procedure NEWMTS, the sets $\rightarrow_\Diamond$, $\rightarrow_\Box$ and $\rightarrow_x$ correspond to *may*, *must* and *exc* of Definition 7 for the current models mts_m and fts_m. All those sets are initially empty. In every iteration, procedure NEWMTS checks for every transition of *fts* if its presence condition is compatible with the current model

mts$_m$ to avoid LTS variants with mutual excluding combinations of transitions. Additionally, adding transitions of *fts* as new may-transitions into *mts*$_m$ results in an increased number of derivable LTS variants. Therefore, we have to ensure that (initially) optional transitions become mandatory if there are new variants derivable *mts*$_m$ which are not included in *fts*. In particular, we have to consider two cases where a newly added optional transition t causes *mts*$_m$ to have more variants than *fts*.

(a) The presence condition of t in *fts* is implied by an aggregated model condition resulting from other transitions processed (either added as optional/mandatory or excluded) in previous iterations of constructing *mts*$_m$. Hence, such logical dependencies between presence conditions of transitions of *fts* must be reflected by setting t mandatory in *mts*$_m$. As those cases might arise whenever a transition modality of the current *mts*$_m$ is adapted (namely before and after the second case), we have to perform a corresponding check twice (see lines 28–29 as well as lines 39–40).

(b) After adding t as optional transition to *mts*$_m$, the set of all optional transitions in *mts*$_m$ including t added in previous iterations yields variants which are not included in *fts* (lines 32–33). Hence, t has to be set to mandatory to exclude those variants from *mts*$_m$. Hence, t has to become mandatory in *mts*$_m$ to ensure the restrictions on LTS variants as imposed in *fts*.

Procedure NEWMTS terminates after having processed every transition of *fts* this way. Hence, it holds that $\mathit{fts}_m = \mathit{fts}$ and thus $\forall \mathit{lts} \preceq \mathit{mts}_m : \mathit{fts} \triangleright \mathit{lts}$.

2. Proof by induction. Initially, $\mathcal{M}$ contains no transitions MTS. In each iteration of procedure MAIN, the blocking clause m specifies exactly those configurations of *fts* not yet being covered by some LTS variant of an MTS in set $\mathcal{M}$. The blocking clause is initialized with the set of all valid configurations of *fts* and is refined after every invocation (including recursive restarts) of *NewMTS* by excluding those configuration being covered by the newly generated *mts* (line 7). Hence, procedure MAIN terminates only after heaving covered every LTS variant of *fts* by at least one MTS in $\mathcal{M}$ and thus $\forall \mathit{lts} \cdot \ \mathit{fts} \triangleright \mathit{lts} : (\exists \mathit{mts} \in \mathcal{M} : \mathit{lts} \preceq \mathit{mts})$. □

We next explore the notion of *minimality* of MTS encodings of FTS in more detail and investigate whether Algorithm 1 is able to generate a minimal MTS encoding.

5.2. Minimality of MTS Encodings

As already mentioned before, there always exists a trivial encoding $\mathcal{M} \in \mathit{context}(fts)$ in which every MTS *mts* $\in \mathcal{M}$ constitutes an LTS such that $|\mathcal{M}| = |\Lambda|$. In some cases, however, this *maximal* solution is also the only valid solution (e.g., example in Figure 4a). Conversely, we intuitively expect a *minimal* solution $\mathcal{M} \in \mathit{context}(fts)$ to consist of a minimum number of MTSs. For instance, assume the transitions of the FTS in Figure 4a to be annotated with two different features f and f', both being optional and independent. A minimal solution would consist of one MTS having both transitions as optional transitions. Now, assume both transitions to be annotated with

the same mandatory feature f. Then, a minimal solution would, again, consist of one MTS having both transitions, but now as mandatory transitions (i.e., being an LTS).

In general, it is not obvious how to characterize an MTS encoding as minimal. Intuitively, we require for an MTS encoding to be minimal that each MTS $\mathcal{M} \in$ *context*(fts) contains as many optional transitions as possible as every optional transition doubles the number of FTS implementations subsumed by a single MTS (i.e., an MTS $m \in \mathcal{M}$ with k may-transitions subsumes 2^k LTS variants). However, simply counting the number of optional transitions may be misleading as the set of LTS variants derivable from two different MTS may be overlapping or even be similar. For instance, when removing the optional transitions from the MTS in Figure 5d both result in the same LTS. Additionally, we should require each pair of MTSs of an MTS encoding to not contain any mutually bisimilar variants.

Another, more technical, issue arises from the possible unrolling of loops: even if the number of MTSs in an MTS interpretation $\mathcal{M} \in$ *context*(*fts*) is minimal, the number of transitions in MTS $mts \in \mathcal{M}$ may be arbitrarily increased as compared to the FTS due to (redundant, yet valid) unrollings of loops. The FTS in Figure 4d and the corresponding MTS in Figure 5d provide an example. Here, the FTS contains three states and several loops between the states s_1 and s_2, whereas the corresponding MTSs both contain five states due to the adaptions of the path conditions throughout the construction steps performed by the algorithm as described above.

To summarize, minimality of MTS encodings may be characterized by lifting the notion of modal refinement to sets of MTS as follows.

Definition 11 (Minimal MTS Encoding)**.** *Let $\mathcal{M}$, $\mathcal{M}'$ be sets of MTSs. By $\mathcal{M}' \sqsubseteq \mathcal{M}$ we denote that*

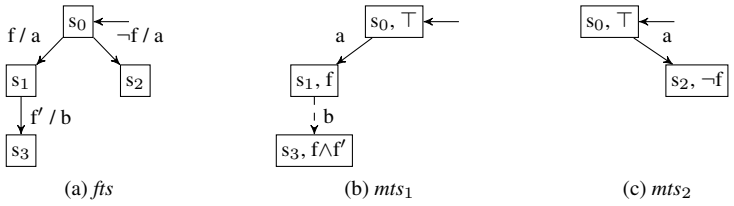
$$\forall mts' \in \mathcal{M}' : \exists mts \in \mathcal{M} : mts' \preceq mts$$

holds. An MTS encoding $\mathcal{M} \in$ context(fts) is minimal for FTS fts iff it is a greatest element of set context(fts) with respect to $\sqsubseteq$ and it holds that

$$\forall mts, mts' \in \mathcal{M} : (mts \preceq mts' \Rightarrow mts = mts').$$

Note, that $\mathcal{M} \in$ *context*(*fts*) is minimal for *fts* iff it is a *greatest* element of *context*(*fts*) thus subsuming a maximum number of (more refined) MTSs and therefore also LTS variants. Furthermore, the first condition of Definition 11 does not imply the second one as the first condition does not forbid having MTSs with bisimilar variants in $\mathcal{M}$ (or $\mathcal{M}'$). Additionally, $\sqsubseteq$ is a preorder on the set *context(fts)* as a minimal MTS encoding of an FTS *fts* is not necessarily unique. Furthermore, for Algorithm 1 to produce minimal MTS encodings, we have to impose a restriction on the corresponding input FTS models, referred to as *structurally deterministic* FTS. In particular, we call an FTS is structurally deterministic if there exists no state having more than one outgoing transitions labeled with the same action, regardless of the (in-)compatibility of their presence conditions.

Definition 12 (Structurally Deterministic FTS)**.** *FTS* $(S, A, F, \rightarrow, \Lambda, p_{init})$ *is* structurally deterministic *if* $\forall t = (p, f, a, p') \in \rightarrow$: $(\forall t' = (p, f', a', p'') \in \rightarrow$: $t \neq t' \Rightarrow a \neq a')$.



(a) *fts* (b) mts_1 (c) mts_2

Figure 11: Example for the Problem of Structurally Non-deterministic FTSs for Algorithm 1

Figure 11 provides an example (with all features being optional) for a structurally non-deterministic FTS to illustrate the problem such an FTS causes to Algorithm 1. In particular, the FTS (cf. Figure 11a) is structurally non-deterministic as state s_0 has two outgoing transitions being labeled with a (although having mutually excluding presence conditions). As a result, we obtain two MTSs mts_1 and mts_2 (cf. Figures 11b and 11c) each with a mandatory transition similarly labeled a which, however, correspond to different FTS transitions with mutually excluding presence conditions. As a consequence, it holds that $mts_2 \preceq mts_1$ and hence the result is not minimal according to Definition 11. To avoid those cases, we require structurally deterministic FTSs for Algorithm 1 to derive a minimal MTS encoding. However, as a future work we plan to extend Algorithm 1 with a post-processing step to check for each generated MTS if it is already covered by another MTS (between lines 5 and 6). We omitted this (presumably very expensive) check in the current version of our tool as it does no occur in any of our subject systems.

Next, we prove that Algorithm 1 derives a minimal MTS encoding for any given structurally deterministic FTS.

Theorem 3. *Let $\mathcal{M} \in context(fts)$ be the MTS encoding of a structurally deterministic FTS fts generated by Algorithm 1. Then $\mathcal{M}$ is the* minimal *MTS encoding of FTS fts.*

Proof. We prove Theorem 3 by showing that (1) $\forall m_i \in \mathcal{M} : \rightarrow^i_\square$ is *minimal*, and (2) $\forall m_i, m_j \in \mathcal{M} : \neg(m_i \preceq m_j \vee m_j \preceq m_i)$, both by contradiction.

1. Let us assume that there exists an $m_i \in \mathcal{M}$ such that $\rightarrow^i_\square$ is not minimal. In this case, there must be a step in Algorithm 1 where an optional transition unnecessarily becomes mandatory. Initially, every transition from *fts* which has not to be excluded from m_i is added as an optional transition. Optional transitions may become mandatory due to two reasons as already shown for Theorem 2.

   (a) The presence condition of a newly added optional transition t is implied by the presence condition of another transition or by the combination of presence conditions of a set of other transitions (lines 28–29). Hence, t must be mandatory as t must be included in every variant in which these other transition(s) are also included.

   (b) Due to the presence condition of the newly added transition t, a combinations of optional transitions (including t) yields an invalid variant (lines

32–33). Hence, t must be mandatory as otherwise the set of generated MTSs contains more variants than the FTS.

In addition, we have to consider those cases in which optional transitions might (unnecessarily) become mandatory due to loop unrolling. However, in these cases, procedure NEWMTS is restarted with a refined blocking clause to avoid unnecessary unrollings (lines 25–26, 30–31, 35–36, and 41–42). As a consequence, Algorithm 1 results in $\rightarrow^{i}_{\Box}$ being minimal.

2. Assume that there exists $m_i, m_j \in \mathcal{M}$ with $m_i \preceq m_j$. Hence, there exists at least one variant v of the product line with $v \preceq m_i$ and $v \preceq m_j$. This is only possible if the FTS is structurally non-deterministic as this would require two bisimilar variants with different feature configurations. For a structurally deterministic FTS, this is avoided by imposing, and iteratively refining, the blocking clause in each new call of procedure NEWMTS (lines 5–7). In this way, any MTS having at least one variant already covered by some previously generated MTS will no more be (re-)generated in any subsequent run. Hence, we have $m_i \not\preceq m_j$.

From (1) and (2) it follows that $\mathcal{M}$ is minimal for structurally deterministic FTSs according to Definition 11. □

6. Implementation and Evaluation

In this section, we present experimental evaluation results gained from applying our approach to a collection of FTS models. To this end, we have implemented Algorithm 1 in a tool which allows us to generate a *minimal MTS encoding* from a given input FTS model as described in the previous section.

6.1. Experimental Setup

The first goal of our evaluation is to investigate general applicability of the algorithm to differing input FTS models. In addition, we are interested in the computational effort for generating a minimal set of MTSs from an FTS as well as the average number of MTSs required for a minimal MTS encoding of an FTS as compared to the maximum number of MTSs (i.e., the number of LTS variants derivable from the FTS). In particular, we consider the following research questions.

Research Questions.

- **RQ1 (Efficiency).** What is the computational effort for generating a minimal set of MTSs for a given FTS, as compared to the maximal set?
- **RQ2 (Effectiveness).** What is the average number of MTSs in a minimal set for a given FTS, as compared to the maximal set?

To address both questions, we applied our tool to a collection of subject systems comprising both case studies from the research community on FTSs as well as synthetically generated FTS models.

Table 1: Subject Systems from this paper (1–5) and Real-World Subject Systems [22] (6–14)

Subject System	# Features	# Variants	# States	# Transitions	# Annotated Transitions	Description
1: Figure 4a	1	2	3	2	2	Figure 4a of this paper
2: Figure 4b	2	3	5	4	4	Figure 4b of this paper
3: Figure 4c	2	3	5	4	2	Figure 4c of this paper
4: Figure 4d	3	6	3	4	4	Figure 4d of this paper
5: Arcade Game Maker	8	8	5	13	13	Running example of this paper
6: Simple Traffic Light	1	1	4	4	0	Simple traffic light loop
7: Complex Traffic Light	1	1	4	5	0	Variant of the simple traffic light loop
8: Hot Drink Machine	9	28	14	21	17	Coffee/tea machine with multiple currencies
9: Sensor Subsystem	7	2	3	15	6	Sensor subsystem of a car wiper system
10: Wiper Subsystem	7	2	5	14	6	Wiper subsystem of a car wiper system
11: Modified Wiper Subsystem	8	4	5	14	7	Wiper subsystem of a car wiper system with permanent wiping
12: Mine Pump Controller	4	4	25	36	35	Controller of a water pumping system
13: Mine Pump System States	1	1	5	18	0	Supplement to the mine pump controller
14: Refined Mine Pump Controller	9	40	25	36	35	Mine pump controller with additional water level readings

Implementation. We implemented Algorithm 1 in a JAVA-tool called MooSE (**Mo**dal Transit**i**on **S**ystem **E**ncoding). As part of our tool, we utilize the SAT solver SAT4J [28] for reasoning about satisfiability of feature constraints. Furthermore, we use the Java Universal Network/Graph (JUNG) framework[1] to create a GUI front-end for easily operating our tool and for visualizing FTS and MTS models. Besides Algorithm 1, our tool further incorporates an automated bisimulation check between FTS and MTS models as described in Section 4 which, for instance, allows the user to verify correctness of generated models.

In order to make our results reproducible, we provide our tool implementation (together with a manual) and our case studies on a supplementary web page[2].

Subject Systems. We applied our experiments to 51 subject systems, where the first group comprises 14 FTSs which are taken from existing case studies initially published in [22] (cf. Table 1) as well as the examples created for this paper. In addition, the other group consists of 37 synthetically generated FTSs. In particular, the first group consists of FTS case studies modeling traffic light controls, a vending machine for coffee and tea, subsystems of a wiper system, and several different parts of mine-pump control systems. The smallest case study, *Simple Traffic Light*, consists of one (mandatory) feature, where the corresponding feature model hence defines one valid configuration and a corresponding LTS variant. The FTS of the *Simple Traffic Light* consists of four

[1] `http://jung.sourceforge.net`
[2] `https://www.es.tu-darmstadt.de/fts2mts/`

states and four transitions from which none is annotated by a presence condition. We added this example to our corpus to check whether our algorithm produces correct results also for such corner cases. The largest case study, *Refined Mine Pump Controller*, has nine features with 40 possible configurations and the respective FTS consists of 25 states and 36 transitions from which 35 are annotated by a presence condition thus constituting variable behavior.

In the second group, we further consider randomly generated FTS models in order to investigate in more detail the impact of different FTS properties on the resulting MTS encoding. We do this by first applying an existing tool for generating feature models and *Featured Finite State Machines* (FFSM) [24] being (non-hierarchical) *Finite States Machines* where transitions are, similar to FTS models, annotated with presence conditions. We translate FFSM to FTS by simply copying the set of states and transitions together with their presence conditions. In contrast, the transition labels on FFSM, which are much more compound than those of FTS, are treated as one atomic action per transition as the actual labeling is not relevant for Algorithm 1. In particular, we generated three FTSs as well as three feature models (i.e., one feature model for each FTS). We then adapted each FTS, e.g., by removing transitions, and each feature model, e.g., by changing an or-group to an alternative group, to obtain a diverse corpus. Here, the case studies vary between twelve to 16 features, 50 to 10 states, 21 to 79 transitions, and five to 50 variants.

Experiment Design and Measurement Setup. In order to evaluate our approach, we generated a minimal MTS encoding as well as the maximal set of MTSs (i.e., the set of all LTS variants of the input FTSs) for each of our subject systems. To answer research question **RQ1**, we measured CPU times required for applying Algorithm 1 as compared to generating all LTS variants from the given FTS models. Concerning **RQ2**, we additionally counted the number of MTS models of the minimal encoding as well as the number of LTS variants of the maximal encoding for the given FTS models. We used SAT4J version 2.3.4, and we applied all experiments on a machine with Windows 10 x64 and 12GB of RAM running on an Intel Xeon E3-1230v3 (4x3.3GHz) processor. We reran the experiments several times and observed that the deviation of the results among the different runs are negligible.

6.2. Results and Discussion

We next present the measurement results of our experiments together with a discussion of the results with respect to our research questions.

Results. The measurement results addressing RQ1 and RQ2 are shown in Figure 12 (real-world case studies) and Figure 13 (synthetic case studies).

- **RQ1 (Efficiency).** The average CPU time required for the real-world case studies is 5.2 s with a geometric mean of 65.9 ms, ranging from less than 1 ms (*Simple Traffic Light*) to 59.7 s (*Refined Mine Pump Controller*). The average CPU time required for the synthetic case studies is 8.1 min with a geometric mean of 170.5 s, ranging from 5.2 s (case study 1 having six variants) to 36.9 min (case study 35 having 20 variants). In contrast, generating the maximal set of MTSs (i.e.,

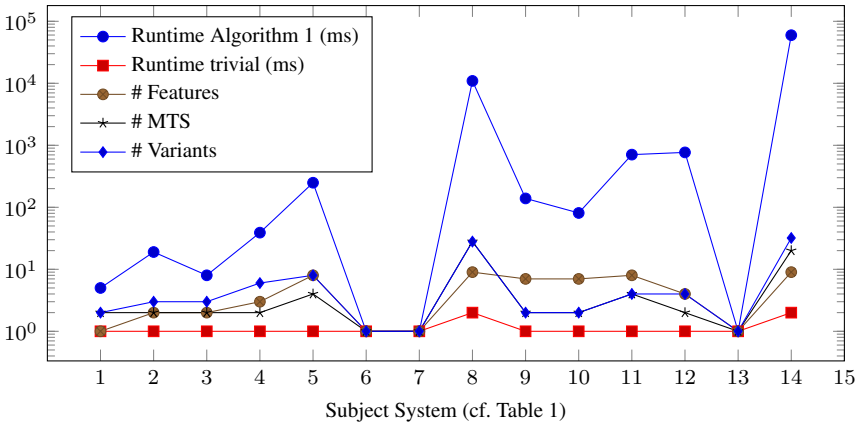


Figure 12: Results for the Real-Word Case Studies (cf. Table 1) with Logarithmic y-Axis

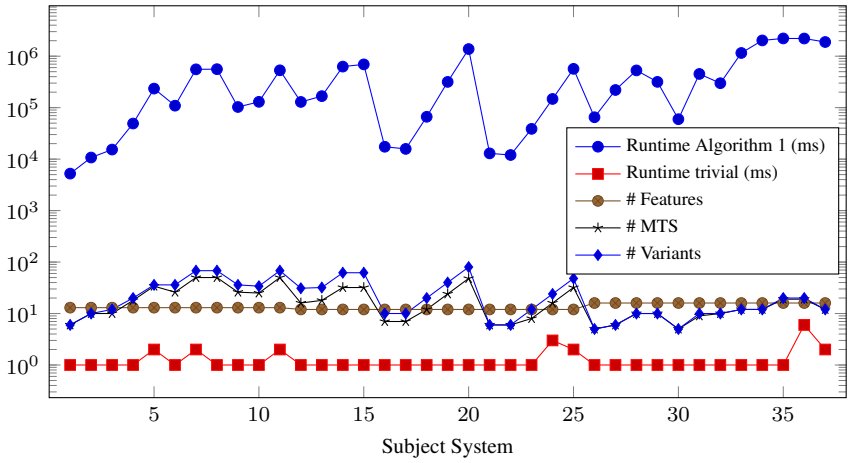


Figure 13: Results for the Synthetic Case Studies with Logarithmic y-Axis

directly deriving the set of LTS variants) takes at most 2 ms for all case studies (except for the synthetic case study 36 taking 6 ms).

- **RQ2 (Effectiveness).** Concerning the real-world case studies, the minimum number of MTSs equals the maximum number of MTSs in some cases (namely 1, 6, 7, 8, 9, 10, 11, and 13) while for other case studies, the minimum number of MTSs is considerably smaller than the maximum number (e.g., case study 14 can be encoded as 20 MTSs comrising 32 LTS variants). Here, the geometric mean is 2.7 for the minimum number of MTSs and 3.5 for the maximum number. Note that the three case studies (6, 7, and 13) each have only one variant to demonstrate that our algorithm indeed does not produce sets of MTSs with overlapping variants. Without these special cases, the geometric mean is 3.6 for the minimum number of MTSs and 5.0 for the maximum number. For the synthetic case studies, Algorithm 1 produces 14.8 MTSs and a maximum 18.6 MTSs, again considering the geometric mean. Again, note that we observe a number of cases where the minimum and maximum number of MTSs coincide (1, 2, 21, 22, 26 to 30, 32, 33, 34, and 37) as no optional transitions can ever be produced in any MTS. For the rest of the synthetic case studies, the number of MTSs ranges from 10 (minimum) to 12 (maximum) and 48 (minimum) to 80 (maximum).

Discussion and Summary. We now discuss the results of our experimental evaluation with respect to the research questions **RQ1** and **RQ2**.

- **RQ1 (Efficiency).** From the results obtained from both the real-world case studies as well as the synthetically generated case studies, we can conclude that there is no obvious correlation between the different structural size measures (e.g., number of states and transitions) of the FTS models and the CPU time consumed by Algorithm 1. Instead, we observe a potential correlation between CPU time and the number of variants for most of the case studies. To summarize, we may conclude that our approach is capable to also scale to FTS models comprising considerably larger sets of variants as the average CPU time is between 5.2 s (real-world) and 8.1 min (synthetic). Even the largest synthetic case study only takes about 37 min. However, it should be noted that generating a minimum set of MTSs is considerably slower than generating the maximum set, taking only 6 ms in the worst case (for the synthetic case study 36).

- **RQ2 (Effectiveness).** Concerning effectiveness of Algorithm 1, there is no obvious correlation between the minimal and maximal solution. As described above, there are structurally small as well as larger case studies for which the number of variants and the number of MTSs, however, is equal (for both real-world and synthetic case studies). Instead, we observe that the number of generated MTSs not only depends on the structure of the FTS but also on the dependencies between features and presence conditions as also illustrated by the different examples in Section 4. For our real-world case studies, the minimum number of MTSs is 23% smaller than the maximum number. When leaving out the special cases 6, 7, and 13 (only consisting of one variant) the minimum number of MTSs is 28% smaller than the maximum number. For the synthetic case studies,

we observe similar results. Here, the minimum number of MTSs is 20% smaller than the maximum number. Additionally, when leaving out the case studies 26 to 37 (where the minimum and maximum are very similar due a high degree of dependencies between presence conditions of different transitions), the minimum number of MTSs is, again, 28% smaller than the maximum number. To summarize, there are no obvious (i.e., syntactic) structural properties of FTS models indicating a clear correlation between the sizes of the minimum and maximum MTS encoding.

6.3. Threats to Validity

We conclude this section with a brief discussion of threats to validity potentially obstructing our evaluation results.

Internal Threats. Concerning the correctness of our approach, we provide a detailed proof in Section 4 showing that the MTS encoding is, up to bisimulation equivalence, both sound and complete with respect to the set of variants represented by the input FTS model. In addition, we prove in Section 5 that the MTS encoding generated by Algorithm 1 constitutes a minimal solution. In this regard, one possible threat to internal validity might arise from the correct implementation of the approach in our tool. To address this issue, we exhaustively tested our tool implementation using a variety of different examples including default cases as well as corner cases. In addition, the only major external component used in our tool is SAT4J which is a mature SAT-solver widely used in practice.

Another potential threat to the internal validity of our evaluation results may arise from the inherent non-determinism of Algorithm 1 concerning, for instance, the ordering in which the state-transition graph is traversed in an iteration. As a result, the resulting minimal MTS encoding is not unique thus leading to different measurement results for different runs with the same input model.

External Threats. One potential threat to external validity might arise from the lack of comparison with other approaches. However, we are not aware of any competitive approach so far in recent literature aiming at generating a minimal MTS encoding as pursued in our approach (see Section 7 for details). However, one interesting path to follow in a future work would be to consider MTS with variability constraints as recently proposed in [11]. As this extension increases expressiveness of MTSs to equal that of FTSs [13], it may permit an even more succinct encoding of FTS as compared to the plain MTSs considered in our setting.

Finally, the selection of subject systems might always threaten external validity. For our experiments, we selected well-known community benchmarks as well as synthetically generated models. However, although we are confident that our collection covers a variety of crucial cases and model sizes, the lack of real-world FTS models might obstruct any generalization of our evaluation results.

7. Related Work

In this section, we discuss related work on relating the semantic models (and therefore comparing the expressiveness of) different modeling formalisms for software product lines from the recent literature. To this end, we limit our considerations to (operational) behavioral-variability modeling formalisms based on (variations of, or extensions to) LTSs as the underlying semantic foundation. The goal of all considered approaches is, in general, to avoid that every possible model variant derivable from a software product line has to be explicitly modeled as a dedicated LTS. Instead, the different approaches propose (syntactic and/or semantic) mechanisms for integrating several model variants into one concise model.

To our knowledge, our approach in encoding FTSs into a set of MTSs, which is semantic preserving, is new. Based on this concept, expressiveness of the different approaches can be characterized by the minimum number of models required to cover all variants. In this regard, FTSs and MTSs (with finite behavior) can be seen as two extrema of an expressiveness spectrum, as one FTS always suffices to comprise all possible LTS variants (but, with the disadvantage of a complex representation), whereas MTSs are inherently limited by only being capable of distinguishing between mandatory and optional transitions (but, with the advantage of a simple representation).

As an alternative way of comparing expressiveness, Beohar et al. have recently proposed to define encodings between formalisms such that a hierarchy of expressiveness is naturally built upon the (non-)existence of (mutual) encodings [12]. Ter Beek et al. have contributed to this expressiveness hierarchy by demonstrating that MTSs with variability constraints are equally expressive as FTSs [13]. In contrast, Benduhn et al. survey different modeling formalisms in terms of their suitability for applying different product-line analysis strategies [29]

Concerning MTSs [2] in particular, Fischbein et al. [30] were the first to argue that these models are adequate for modeling behavioral variability in software product lines. Thereupon, several researchers used MTSs as well-suited formalism to perform rigorous analysis of software product lines [4, 5, 6, 31, 7]. In order to cope with the limited expressive power of MTSs and to further restrict the set of valid model variants derivable from an MTS, various approaches combine MTSs with additional constraints expressed in a deontic logic called Modal-Hennessy-Milner-Logic (MHML) [4, 5, 6] as well as so-called variability constraints [11]. Furthermore, Benes et al. in [27], introduce an extension of MTSs with a set of parameters and define obligation functions on the set of atomic propositions, which are related to each state and contain transitions emanating the state and parameters. By setting different valuations for the parameters and also using different atomic propositions the presence or absence of transitions can be specified. Using this formalism global/persistent choices can be made throughout a model. Křetínský and Sickert [32] show that this extension of MTS may be translated to Boolean MTS, whereas they do not consider a translation into (sets of) plain MTSs as done in our work. Basile et al. in [33], introduce an extension of contract automata with modality [34], in which necessary requests are distinguished from permitted requests, with feature constraints. These models can be used for modeling the behavior of contract-based dynamic service product lines. Other approaches exploit principles from interface theories to restrict the set of derivable variants from MTS

to only those being compatible under parallel composition to a given environmental specification [7, 31]. In [9], two variants of MTSs, namely disjunctive modal transition systems (DMTSs) [35] and 1MTSs are compared from the expressiveness point of view. DMTSs are similar to 1MTSs in that both rely on the notion *hyper* transitions, in order to explicitly relate transitions in MTSs to (sets of) features of a product line. The difference is in the interpretation of such transitions. In DTMSs, must-hyper-transitions represent an or-relation between multiple choices, whereas this restriction is not made in 1MTS. In [9], it is shown that both formalisms have the same expressive power, i.e., they induce the same sets of LTSs as their implementations.

Concerning FTSs, as initially proposed by Classen et al. [36], those models have been mostly utilized for efficient temporal model-checking of entire product lines by solely considering one FTS model [1]. Thereupon, Cordy et al. [37] extended this earlier work by combining non-Boolean features and multi-features in a high-level specification language called TVL*. An algorithm for constructing an FTS from a behavioral specification written in TVL* was also given.

Finally, PL-CCS [38], introduced by Gruler et al. [38], constitutes an extension of Milner's CCS [39] by means of an alternative choice operator called "binary variant" to choose (and memorize) behavioral variations in CCS step semantics. Similar to MTSs, the validity of variants can be further restricted using the multi-valued modal μ-calculus [40].

To summarize, none of these existing approaches for comparing product-line modeling formalisms yet followed the idea as proposed in this paper, by encoding a class of models, FTSs, into another class of models, MTSs, requiring *sets of models*, where for finite behavior FTSs are more expressive than MTSs, such that both comprise equivalent sets of variants.

8. Conclusions and Future Work

In this paper, we presented an encoding of FTSs into sets of MTSs with an equivalent set of LTS variants. We also gave an algorithmic interpretation of this translation and proved it to be correct. Moreover, we discussed the issue of minimality of the computed translations and proved a particular notion of minimality for the outputs of our algorithm.

The concept developed in this paper allows for a novel assessment concerning the expressiveness of variability-modeling formalism in terms of the number of models required for covering all variants. Based on this new concept, we aim at defining new encoding and expressiveness criteria as future work. As a result, we are targeting the definition of a dense spectrum of variability-modeling formalisms, having FTSs and MTSs (or, plain LTSs, respectively) as its extrema. In this regard, also the possible influence of potentially infinite sets of states of these formalisms on expressiveness shall be taken into account. Furthermore, we plan to adapt the algorithm such that the requirement of structurally deterministic FTSs may be dropped to obtain minimal MTS encodings.

Acknowledgment. The work of Mahsa Varshosaz and Mohammad Reza Mousavi has been partially supported by grants from the Swedish Knowledge Foundation (Stiftelsen

for Kunskaps- och Kompetensutveckling) in the context of the AUTO-CAAS HoGproject (number: 20140312), Swedish Research Council (Vetenskapsradet) award number: 621-2014-5057 (EffectiveModel-Based Testing of Concurrent Systems), and the ELLIIT Strategic Research Environment. The work of Lars Luthmann, Paul Mohr and Malte Lochau has been supported by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future – Managed Software Evolution (LO 2198/2-1).

References

[1] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, J.-F. Raskin, Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking, IEEE Transactions on Software Engineering 39 (8) (2013) 1069–1089. doi:10.1109/TSE.2012.86.

[2] K. Larsen, B. Thomsen, A modal process logic, in: Proc. of the 3rd Annual Symposium on Logic in Computer Science (LICS '88), IEEE, 1988, pp. 203–210.

[3] K. G. Larsen, L. Xinxin, Equation solving using modal transition systems, in: Proceedings of ACM/IEEE Symposium on Logic in Computer Science (LICS 1990), IEEE Computer Society, 1990, pp. 108–117.

[4] P. Asirelli, M. H. ter Beek, S. Gnesi, A. Fantechi, Formal description of variability in product families, in: Proceedings of the 15th International Software Product Line Conference (SPLC '11), IEEE, 2011, pp. 130–139.

[5] P. Asirelli, M. H. ter Beek, A. Fantechi, S. Gnesi, A model-checking tool for families of services, in: Proc. of the International Conference on Formal techniques for distributed systems (FMOODS'11/FORTE'11), Vol. 6722 of Lecture Notes in Computer Science, Springer, 2011, pp. 44–58.

[6] P. Asirelli, M. H. ter Beek, A. Fantechi, S. Gnesi, A compositional framework to derive product line behavioural descriptions, in: Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISoLA '12), Vol. 7609 of Lecture Notes in Computer Science, Springer, 2012, pp. 146–161.

[7] K. G. Larsen, U. Nyman, A. Wąsowski, Modal I/O automata for interface and product line theories, in: R. D. Nicola (Ed.), Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP'07), Vol. 4421 of Lecture Notes in Computer Science, Springer, 2007, pp. 64–79.

[8] M. H. ter Beek, A. Lluch-Lafuente, M. Petrocchi, Combining declarative and procedural views in the specification and analysis of product families, in: Proceedings of the 17th International Software Product Line Conference co-located workshops (SPLC '13 workshops), ACM, 2013, pp. 10–17.

[9] H. Fecher, H. Schmidt, Comparing disjunctive modal transition systems with an one-selecting variant, The Journal of Logic and Algebraic Programming 77 (1-2) (2008) 20–39. doi:http://dx.doi.org/10.1016/j.jlap.2008.05.003.

[10] J. Křetínsky, Modal transition systems: Extensions and analysis, Ph.D. thesis, Masaryk University (2014).

[11] M. H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints, Journal of Logical and Algebraic Methods in Programming 85 (2) (2016) 287 – 315.

[12] H. Beohar, M. Varshosaz, M. R. Mousavi, Basic behavioral models for software product lines: Expressiveness and testing pre-orders, Sci. Comput. Program. 123 (2016) 42–60. doi:10.1016/j.scico.2015.06.005.
URL https://doi.org/10.1016/j.scico.2015.06.005

[13] M. H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, L. Paolini, On the expressiveness of modal transition systems with variability constraints, Science of Computer Programming 169 (2019) 1 – 17. doi:https://doi.org/10.1016/j.scico.2018.09.006.

[14] M. Varshosaz, M. R. Mousavi, Comparative expressiveness of product line calculus of communicating systems and 1-selecting modal transition systems, in: Theory and Practice of Computer Science - In Proceedings of 45th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2019, Nový Smokovec, Slovakia, January 27-30, 2019, 2019, pp. 490–503. doi:10.1007/978-3-030-10801-4_38.
URL https://doi.org/10.1007/978-3-030-10801-4_38

[15] M. Varshosaz, H. Beohar, M. R. Mousavi, Basic behavioral models for software product lines: Revisited, Science of Computer Programming 168 (2018) 171 – 185. doi:https://doi.org/10.1016/j.scico.2018.09.001.
URL http://www.sciencedirect.com/science/article/pii/S0167642318303381

[16] N. D'Ippolito, D. Fischbein, M. Chechik, S. Uchitel, MTSA: The modal transition system analyser, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), 2008, pp. 475–476.

[17] G. Verdier, J.-B. Raclet, MAccS: a tool for reachability by design, in: International Workshop on Formal Aspects of Component Software, Springer, 2014, pp. 191–197.

[18] J. Křetínskỳ, S. Sickert, MoTraS: A tool for modal transition systems and their extensions, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2013, pp. 487–491.

[19] S. S. Bauer, P. Mayer, A. Legay, MIO workbench: A tool for compositional design with modal input/output interfaces, in: Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA 2011), Vol. 6996 of Lecture Notes in Computer Science, Springer, 2011, pp. 418–421.

[20] M. H. ter Beek, F. Mazzanti, A. Sulova, VMC: A tool for product variability analysis, in: D. Giannakopoulou, D. Méry (Eds.), FM 2012: Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 450–454.

[21] M. H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, L. Paolini, From featured transition systems to modal transition systems with variability constraints, in: Proceedings of Software Engineering and Formal Methods (SEFM'17), Vol. 9276 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 344–359.

[22] A. Classen, Modelling with FTS: a collection of illustrative examples, Tech. Rep. P-CS-TR SPLMC-00000001, University of Namur, available online at `https://pure.fundp.ac.be/ws/files/1051983/69416.pdf` (2010).

[23] SEI: A framework for software product line practice, `http://www.sei.cmu.edu/productlines/tools/framework/`, accessed: 2017-03-07.

[24] V. Hafemann Fragal, A. Simao, M. R. Mousavi, Validated test models for software product lines: Featured finite state machines, in: O. Kouchnarenko, R. Khosravi (Eds.), Revised selected papers of the 13th International Conference on Formal Aspects of Component Software (FACS'16), Vol. 10231, Springer International Publishing, Cham, 2017, pp. 210–227.

[25] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990).

[26] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[27] N. Beneš, J. Křetínský, K. G. Larsen, M. H. Møller, J. Srba, Parametric modal transition systems, in: T. Bultan, P.-A. Hsiung (Eds.), Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11), Vol. 6996 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2011, pp. 275–289.

[28] D. Le Berre, A. Parrain, The SAT4J library, Release 2.2, JSAT 7 (2-3).

[29] F. Benduhn, T. Thüm, M. Lochau, T. Leich, G. Saake, A survey on modeling techniques for formal behavioral verification of software product lines, in: Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, (VaMoS'15), ACM, New York, 2015, pp. 80–87.

[30] D. Fischbein, S. Uchitel, V. Braberman, A foundation for behavioural conformance in software product line architectures, in: Proceedings of the ISSTA Workshop on Role of software architecture for testing and analysis (ROSATEA'06), ACM, 2006, pp. 39–48.

[31] M. Lochau, J. Kamischke, Parameterized preorder relations for model-based testing of software product lines, in: Proceedings of the 5th Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISOLA' 12), Vol. 7609 of Lecture Notes in Computer Science, Springer, 2012, pp. 223–237.

[32] J. Křetínský, S. Sickert, On refinements of Boolean and parametric modal transition systems, in: Z. Liu, J. Woodcock, H. Zhu (Eds.), Proceedings of the 10th International Colloquium on Theoretical Aspects of Computing (ICTAC 2013), Springer, 2013, pp. 213–230.

[33] D. Basile, M. H. ter Beek, F. Di Giandomenico, S. Gnesi, Orchestration of dynamic service product lines with featured modal contract automata, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, SPLC '17, ACM, New York, NY, USA, 2017, pp. 117–122. doi:10.1145/3109729.3109741.
URL http://doi.acm.org/10.1145/3109729.3109741

[34] D. Basile, F. Di Giandomenico, S. Gnesi, P. Degano, G.-L. Ferrari, Specifying variability in service contracts, in: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS '17, ACM, New York, NY, USA, 2017, pp. 20–27. doi:10.1145/3023956.3023965.
URL http://doi.acm.org/10.1145/3023956.3023965

[35] K. G. Larsen, L. Xinxin, Equation solving using modal transition systems, in: Proc. of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), IEEE Computer Society, 1990, pp. 108–117. doi:10.1109/LICS.1990.113738.
URL http://dx.doi.org/10.1109/LICS.1990.113738

[36] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, J.-F. Raskin, Model checking lots of systems: efficient verification of temporal properties in software product lines, in: Proceedings of the 32nd International Conference on Software Engineering (ICSE '10), Vol. 1, ACM, 2010, pp. 335–344.

[37] M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, Beyond Boolean product-line model checking: dealing with feature attributes and multi-features, in: D. Notkin, B. H. C. Cheng, K. Pohl (Eds.), Proceedings of the 35th International Conference on Software Engineering (ICSE '13), IEEE / ACM, 2013, pp. 472–481.

[38] A. Gruler, M. Leucker, K. Scheidemann, Modeling and model checking software product lines, in: Proceedings of the Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '08), Vol. 5051 of Lecture Notes in Computer Science, Springer, 2008, pp. 113–131.

[39] R. Milner, A Calculus of Communicating Systems, Vol. 92 of Lecture Notes in Computer Science, Springer, 1982.

[40] S. Shoham, O. Grumberg, Multi-valued model checking games, J. Comput. Syst. Sci. 78 (2) (2012) 414–429.

Appendix A.

Lemma 1. *Considering the definition of the function $const()$, given in Definition 7, this function always has a maximal fixed point.*

Proof. Consider an FTS $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$; given $mts{=}(Q, A, \rightarrow_{\Diamond}, \rightarrow_{\Box}, q_{init})$, where $Q = \mathbb{P} \times \mathbb{B}(F)$; we prove $const((p, e))$, for $(p, e) \in Q$, always has a maximal fixed point as follows. We prove this function is monotone and hence has a fixed point. To this end, we show $const_i((p, e)) \implies const_{i-1}((p, e))$ by applying induction on the index of the function. In the base case for any $(p, e) \in Q$ we have:

$$
\begin{aligned}
&const_0((p,e)) = e \\
&const_1((p,e)) = e \wedge \bigwedge_{(p,a,f,p') \in must((p,e))} \big(const_0((p', e \wedge f))\big) \wedge \\
&\qquad\qquad \bigwedge_{(p,a,f,p') \in may((p,e))} \big(\neg f \vee (const_0((p', e \wedge f)))\big) \wedge \\
&\qquad\qquad \bigwedge_{(p,a,f,p') \in exc((p,e))} \neg f
\end{aligned}
$$

Thus, it holds $const_1((p, e)) \implies const_0((p, e))$.

In the inductive step we consider: for any $(p, e) \in Q$, $\forall_{j \leq i-1}\ const_j((p, e)) \implies const_{i-2}((p, e))$. Then, we prove $const_i((p, e)) \implies const_{i-1}((p, e))$ as well. Based on the above definition:

$$
\begin{aligned}
const_i((p,e)) = e \wedge &\bigwedge_{(p,a,f,p') \in must((p,e))} \big(const_{i-1}((p', e \wedge f))\big) \wedge \\
&\bigwedge_{(p,a,f,p') \in may((p,e))} \big(\neg f \vee (const_{i-1}((p', e \wedge f)))\big) \wedge \\
&\bigwedge_{(p,a,f,p') \in exc((p,e))} \neg f
\end{aligned}
$$

and also it holds:

$$
\begin{aligned}
const_{i-1}((p,e)) = e \wedge &\bigwedge_{(p,a,f,p') \in must((p,e))} \big(const_{i-2}((p', e \wedge f))\big) \wedge \\
&\bigwedge_{(p,a,f,p') \in may((p,e))} \big(\neg f \vee (const_{i-2}((p', e \wedge f)))\big) \wedge \\
&\bigwedge_{(p,a,f,p') \in exc((p,e))} \neg f
\end{aligned}
$$

Given the premise in the inductive step it holds: $\forall_{(p',e')\in Q}\ const_{i-1}((p',e')) \implies const_{i-2}((p',e'))$. Hence, it holds:
$\bigwedge_{(p,a,f,p')\in must((p,e))} \big(const_{i-1}((p',e\wedge f))\big) \implies \bigwedge_{(p,a,f,p')\in must((p,e))} \big(const_{i-2}((p',e\wedge f))\big)$ and $\bigwedge_{(p,a,f,p')\in may((p,e))} \big(\neg f\vee(const_{i-1}((p',e\wedge f)))\big) \implies \bigwedge_{(p,a,f,p')\in may((p,e))} \big(\neg f\vee(const_{i-2}((p',e\wedge f)))\big)$.
Hence, it holds $const_i((p,e)) \implies const_{i-1}((p,e))$.

□

Lemma 2. *Consider an arbitrary fts* $= (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$ *and a set of MTSs* $\mathcal{M} \in context(fts)$. *Consider* $mts=(Q,A,\rightarrow_\Diamond,\rightarrow_\Box,q_{init})$ *s.t.* $mts \in \mathcal{M}$ *and* $\lambda \in \Lambda_{mts}$, *it holds:*

$$\big(\forall_{(p,e)\in Q}\ \lambda \models e\ \wedge\ \lambda \models f\big) \Rightarrow \big(\forall_{(p,f,a,p')\in\rightarrow}\ ((p,e),a,(p',e\wedge f)) \in \rightarrow_\Diamond\ \big)$$

Proof. Based on item 4.(a) in Definition 8, there exists a path in the set of finite paths of mts such as $\rho : q_{init}\ a_0\ (p_1,e_1)\ a_1\ \cdots (p_{n-1},e_{n-1})\ a_{n-1}\ (p,e)$, in which $\forall_{1\leq i\leq n-1}\ e_i = e_{i-1}\wedge f_i$, and $e = e_{n-1}\wedge f_{n-1}$. Based on Definition 8, in each transition $(p,e) \xrightarrow{a} (p',e')$, it holds $e' \implies e$ and as $\lambda \models e$, it can be concluded that $\forall_{1\leq i\leq n-1}\ \lambda \models e_i$, and also as $q_{init} = (p_{init}, \bigvee_{\lambda\in\Lambda}\lambda)$ it holds that $\lambda \models \bigvee_{\lambda\in\Lambda}\lambda$. Hence, the premise of the above implication holds for all the states in ρ, that is an initial path that ends in (p,e). Next, we use induction through the path to prove the above implication holds. Since $\lambda \in \Lambda_{mts}$, then $\lambda \implies const(q_{init})$. Assume that the iterations for computing the function $const$ are fixed in k steps that is $const_k(q_{init}) = const_{k-1}(q_{init})$ (a fixed point exists according to Lemma 1).

We consider the base step of induction:

$$const_k(q_{init}) = e\ \wedge \bigwedge_{(p_{init},a,f,p')\in must((p,e))} \big(const_{k-1}((p',e\wedge f))\big)\ \wedge \bigwedge_{(p_{init},a,f,p')\in may((p,e))} \big(\neg f\vee(const_{k-1}((p',e\wedge f)))\big)\ \wedge \bigwedge_{(p_{init},a,f,p')\in exc((p,e))} \neg f$$

Based on item 4.(a) in Definition 8, $e_1 = \bigvee_{\lambda\in\Lambda}\lambda\wedge f_0$. Since, $\lambda \implies const_k(q_{init})$, and $\lambda \models e_1$, from the above formula it can be concluded that $\forall_{(q_{init},f_0,a,p')\in\rightarrow}$ $(q_{init},a,(p',e_1)) \in \rightarrow_\Diamond$. Otherwise $\neg f_0$, is considered as one of the conjunctions in construction of $const_k((p,e))$, and as $\lambda \models const_k(q_{init})$ then $\lambda \models \neg f_0$, which contradicts $\lambda \models f_0$.

Next, we assume that the implication holds for step $n-1$ in induction.

$$const_{k-n+1}((p_{n-1},e_{n-1})) = e \wedge \bigwedge_{(p_{n-1},a,f,p')\in must((p,e))} \big(const_{k-n}((p',e\wedge f))\big) \wedge \bigwedge_{(p_{n-1},a,f,p')\in may((p,e))} \big(\neg f \vee (const_{k-n}((p',e\wedge f)))\big) \wedge \bigwedge_{(p_{n-1},a,f,p')\in exc((p,e))} \neg f$$

As $e = e_{n-1} \wedge f_{n-1}$ and since $\lambda \models e_{n-1}$ hence, $\lambda \models f_{n-1}$. Since, we assume that the above implication holds in this step of induction it can be concluded that $\exists((p_{n-1},e_{n-1}),a,(p,e)) \in \rightarrow_{\Diamond}$. Hence, given that $\lambda \models f_{n-1}$, and according to the construction of $const_{k-n+1}((p_{n-1},e_{n-1}))$ given above it can be seen that $\lambda \implies const_{k-n}((p,e))$.

Next, we prove that the implication holds in step n. According to the previous case it holds $\lambda \implies const_{k-n}((p,e))$.

$$const_{k-n}((p,e)) = e \wedge \bigwedge_{(p,a,f,p')\in must((p,e))} \big(const_{k-n-1}((p',e\wedge f))\big) \wedge \bigwedge_{(p,a,f,p')\in may((p,e))} \big(\neg f \vee (const_{k-n-1}((p',e\wedge f)))\big) \wedge \bigwedge_{(p,a,f,p')\in exc((p,e))} \neg f$$

As $\lambda \implies f$ and $\lambda \implies const_{k-n}((p,e))$, it can be concluded that $\forall_{(p,f,a,p')\in\rightarrow}$ $((p,e),a,(p',e\wedge f)) \in \rightarrow_{\Diamond}$. Otherwise, $\neg f$, is considered as one of the conjunctions in construction of $const_{k-n}((p,e))$, and as $\lambda \models const_{k-n}((p,e))$ then $\lambda \models \neg f$, which contradicts $\lambda \models f$. □

Appendix B.

As mentioned in Section 3, each FTS represents the behavior of all products in a product line as a whole. Labeled Transition Systems (LTSs) have been used as the underlying semantic domain for FTSs. The behavior of an individual product can be represented as an LTS, which is induced from the FTS. Beohar et al. in [12], provide a product-derivation relation to specify valid LTSs implementing an FTS. Moreover, Classen et al. in [1] have provided the definition of a project operator for deriving the LTSs corresponding to the products of a product line. The former one is a semantical definition of product derivation while the latter has more syntactic essence. In this appendix, we show that for considered FTSs the set of LTSs induced by each of these definitions are equal modulo bisimilarity. This is presented in Theorem 4 and its proof. In the following, first, we give the definition of some of the constructs used in the rest of

this appendix (the rest is included in Section 3). The definition of the project operator given by Classen et al. in [1] is as follows.

Definition 13. *Consider an FTS* $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$*, and a product* $\lambda \in \Lambda$*. The projection of fts on* λ*, denoted by* $fts|_\lambda$*, is an LTS* $(\mathbb{P}, A, \rightarrow', p_{init})$*, where* $\rightarrow' = \{(P, a, Q) \mid \exists (P, a, \phi, Q) \in \rightarrow \cdot\ \phi \models \lambda\}$

Based on the definition of the project operator, some of the transitions of an FTS may be eliminated in the resulting LTS. Hence, a subset of the states in the LTS may only be reachable from the initial state. We define an auxiliary function to find the reachable states from the intial state of an LTS resulted after projection. Assuming that $fts|_\lambda = (\mathbb{P}, A, \rightarrow', p_{init})$; we define the set of states reachable from a state $P \in \mathbb{P}$, denoted by $Reach(P)$, such that $P \in Reach(P)$ and $\forall Q' \in Reach(P)\ \exists (Q, a, \phi, Q') \in \rightarrow' \cdot\ Q \in Reach(P)$.

Finally, the definition of bisimulation for LTSs is as follows.

Definition 14. *Assume* $lts = (\mathbb{S}, A, \rightarrow, s_{init})$ *and* $lts' = (\mathbb{S}', A', \rightarrow', s'_{init})$*. We say* $lts \sim lts'$*, iff there exists a relation* $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}'$ *that holds the following conditions (cf. Definition 7.1. in [26]):*

(α*)* $(s_{init}, s'_{init}) \in \mathcal{R}$*.*

(β*)* $\forall (s, s') \in \mathcal{R}$ *the following holds:*

(1) For all $(s, a, t) \in \rightarrow$ *there exists* $(s', a, t') \in \rightarrow'$ *such that* $(t, t') \in \mathcal{R}$*,*

(2) For all $(s', a, t') \in \rightarrow'$ *there exists* $(s, a, t) \in \rightarrow$ *such that* $(t, t') \in \mathcal{R}$*.*

Next, we show that the set of LTSs derived from an FTS using the above given definitions is the same (up to bisimilarity).

Theorem 4. *For each FTS fts the set of LTSs induced by applying the project operator given in Definition 13 and the set of LTSs that are valid implementations of fts based on Definition 6, are equal modulo bisimilarity.*

Proof. To prove the above theorem, we consider the following two obligations:

- **Obligation 1:** For an FTS fts, $\forall lts \in \mathbb{LTS} \cdot\ lts \triangleright fts\ \exists \lambda \in \Lambda \cdot\ lts \sim fts|_\lambda$.

  Consider an arbitrary $lts = (\mathbb{S}, A, \rightarrow, s_{init})$ and an FTS $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$ such that $lts \triangleright fts$. Based on Definition 6, it holds there exists $\lambda \in \Lambda$ and $\mathcal{R}_\lambda \subseteq \mathbb{P} \times \mathbb{S}$ such that $(p_{init}, s_{init}) \in \mathcal{R}_\lambda$ and $\mathcal{R}_\lambda$ satisfies the following transfer properties.

  (a) $\forall_{P,Q,a,s,\phi} \left(P\ \mathcal{R}_\lambda\ s\ \wedge\ P \xrightarrow{\phi/a} Q \wedge \lambda \models \phi\right)\ \Rightarrow\ \exists_t \cdot\ s \xrightarrow{a} t\ \wedge\ Q\ \mathcal{R}_\lambda\ t$;

  (b) $\forall_{P,a,s,t} \left(P\ \mathcal{R}_\lambda\ s\ \wedge\ s \xrightarrow{a} t\right)\ \Rightarrow\ \exists_{Q,\phi} \cdot\ P \xrightarrow{\phi/a} Q\ \wedge\ \lambda \models \phi \wedge Q\ \mathcal{R}_\lambda\ t$.

Consider the LTS resulting from projection of fts on λ that is $fts|_\lambda = (\mathbb{P}, A, \rightarrow', p_{init})$. (Based on Definition 13 the set of states in an LTS resulted from projecting an FTS on a product configuration remains the same. To avoid confusion in the rest of the proof we use $\mathbf{P}$ to denote the state in the LTS (resulted from projection) that corresponds to the same state P in the FTS.) We show $lts \sim fts|_\lambda$. To this end, we define a relation $\mathcal{R}$ such that $\forall \mathbf{P} \in \mathbb{P}, s \in \mathbb{S} \cdot \ (\mathbf{P}, s) \in \mathcal{R} \Leftrightarrow (P, s) \in \mathcal{R}_\lambda \ \wedge \ \mathbf{P} \in Reach(\mathbf{p_{init}})$.

Next, we show $\mathcal{R}$ is a bisimulation relation. As $(p_{init}, s_{init}) \in \mathcal{R}_\lambda$ and $\mathbf{p_{init}} \in Reach(\mathbf{p_{init}})$ it holds $(\mathbf{p_{init}}, s_{init}) \in \mathcal{R}$. First, we show $\mathcal{R}$ satisfies condition (β).(1) in Definition 14. Consider an arbitrary pair $(\mathbf{P}, s) \in \mathcal{R}$; based on Definition 13:

$$\forall (\mathbf{P}, a, \mathbf{Q}) \in \rightarrow' \ \exists (P, a, \phi, Q) \in \rightarrow \cdot \ \phi \models \lambda \ \ (1)$$

Furthermore, as $(P, s) \in \mathcal{R}_\lambda$, based on (a) it holds:

$$P \xrightarrow{\phi/a} Q \ \wedge \ \phi \models \lambda \Rightarrow \exists t \in \mathbb{S} \cdot \ s \xrightarrow{a} t \ \wedge \ (Q, s') \in \mathcal{R}_\lambda \ \ (2)$$

As $\mathbf{P} \in Reach(\mathbf{p_{init}})$ and $(P, a, Q) \in \rightarrow'$ it holds $\mathbf{Q} \in Reach(\mathbf{p_{init}})$. Hence, $(Q, s') \in \mathcal{R}_\lambda$ (3).

From (1), (2), and (3) it can be concluded that $(\mathbf{Q}, s') \in \mathcal{R}$. Finally, we conclude that for each $(\mathbf{P}, s) \in \mathcal{R}$ it holds:

$$\mathbf{P} \xrightarrow{a}{}' \mathbf{Q} \Rightarrow s \xrightarrow{a} t \ \wedge \ (\mathbf{Q}, t) \in \mathcal{R} \ \ (i)$$

Next, we show that $\mathcal{R}$ satisfies condition (β).(2) in Definition 14.

Considering an arbitrary pair $(\mathbf{P}, s) \in \mathcal{R}$ and a transition $(s, a, t) \in \rightarrow$; based on (b) it holds:

$$\exists Q, \phi \cdot \ P \xrightarrow{\phi/a} Q \ \wedge \ \phi \models \lambda \ \wedge \ (Q, t) \in \mathcal{R}_\lambda (4)$$

For $P \xrightarrow{\phi/a} Q$ where $\phi \models \lambda$, based on Definition 13, it holds $\mathbf{P} \xrightarrow{a}{}' \mathbf{Q}$ (5).

Furthermore, as $(\mathbf{P}, s) \in \mathcal{R}$ then $\mathbf{P} \in Reach(\mathbf{p_{init}})$ and consequently as $P \xrightarrow{a}{}' Q$ it holds $\mathbf{Q} \in Reach(\mathbf{p_{init}})$. Hence, $(\mathbf{Q}, t) \in \mathcal{R}$ (6).

Thus, from (4), (5), and (6) we can conclude that for each arbitrary pair $(\mathbf{P}, s) \in \mathcal{R}$ it holds:

$$s \xrightarrow{a} s' \Rightarrow \mathbf{P} \xrightarrow{a} \mathbf{Q} \ \wedge \ (\mathbf{Q}, t) \in \mathcal{R} \ \ (ii)$$

From (i), (ii), and $(\mathbf{p_{init}}, s_{init}) \in \mathcal{R}$, given Definition 14, we conclude that $\mathcal{R}$ is a bisimulation relation and that $lts \sim fts|_\lambda$.

- **Obligation 2:** For an FTS fts, $\forall \lambda \in \Lambda\ \exists lts \in \mathbb{LTS} \cdot\ lts \triangleright fts\ \wedge\ fts|_\lambda \sim lts$.

  Consider FTS $fts = (\mathbb{P}, A, F, \rightarrow, \Lambda, p_{init})$; for $\lambda \in \Lambda$ we assume $fts|_\lambda = (\mathbb{P}, A, \rightarrow', p_{init})$. (Same as before, to avoid confusion we use $\mathbf{P}$ to denote the state in the LTS, resulted from projection, which corresponds to the same state P in the FTS.) We show that the identical LTS (as $fts|_\lambda$) implements fts considering the product-derivation relation given in Definition 6. To this end, we define a relation $\mathcal{R} \subseteq \mathbb{P} \times Reach(p_{init})$, such that $\forall \mathbf{P} \in Reach(\mathbf{p}_{init}) \cdot (P, \mathbf{P}) \in \mathcal{R}$.

  As $\mathbf{p}_{init} \in Reach(\mathbf{p}_{init})$ it holds $(p_{init}, \mathbf{p}_{init}) \in \mathcal{R}$. Next, we show that $\mathcal{R}$ satisfies the two transfer properties given in Defintion 6.

  Consider a pair $(P, \mathbf{P}) \in \mathcal{R}$: for a transition $P \xrightarrow{\phi/a} Q$ where $\phi \models \lambda$, based on Definition 13, it holds $(\mathbf{P}, a, \mathbf{Q}) \in \rightarrow'$ (1).

  As $(\mathbf{P}, a, \mathbf{Q}) \in \rightarrow'$ and $\mathbf{P} \in Reach(\mathbf{p_{init}})$ then $\mathbf{Q} \in Reach(\mathbf{p_{init}})$. Thus, it holds $(Q, \mathbf{Q}) \in \mathcal{R}$ (2).

  Given (1) and (2), it holds:

$$\forall (P, \mathbf{P}) \in \mathcal{R}, Q \in \mathbb{P}, a \in A, \phi \in \mathbb{B}(F) \cdot\ P \xrightarrow{\phi/a} Q\ \wedge\ \phi \models \lambda \Rightarrow \mathbf{P} \xrightarrow{a} \mathbf{Q}\ \wedge$$

$$(Q, \mathbf{Q}) \in \mathcal{R}\ \ (iii)$$

  Which is the fist transfer property in Definition 6. Next, we prove $\mathcal{R}$ satisfies the second transfer property in the definition.

  Consider a pair $(P, \mathbf{P}) \in \mathcal{R}$; for each transition $\mathbf{P} \xrightarrow{a}{}' \mathbf{Q}$, based on Definition 13, it holds $\exists (P, a, \phi, Q) \in \rightarrow \cdot\ \phi \models \lambda$ (3).

  As $\mathbf{P} \in Reach(\mathbf{p_{init}})$, and $\mathbf{P} \xrightarrow{a}{}' \mathbf{Q}$ it holds $\mathbf{Q} \in Reach(\mathbf{p_{init}})$. Thus, $(Q, \mathbf{Q}) \in \mathcal{R}$ (4).

  Hence, from (3) and (4) we conclude that:

$$\forall (P, \mathbf{P}) \in \mathcal{R}, \mathbf{Q} \in \mathbb{P}, a \in A \cdot\ \mathbf{P} \xrightarrow{a} \mathbf{Q} \Rightarrow \exists \phi \in \mathbb{B}(F) \cdot\ P \xrightarrow{\phi/a} Q\ \wedge\ \phi \models \lambda\ \wedge$$

$$(Q, \mathbf{Q}) \in \mathcal{R}\ \ (iv)$$

  From (iii), (iv), and $(p_{init}, \mathbf{p}_{init}) \in \mathcal{R}$, based on Definition 6, we conclude that $\mathcal{R}$ is a product-derivation relation. Hence, it holds $\forall \lambda \in \Lambda\ \exists lts \in \mathbb{LTS} \cdot\ lts \triangleright fts\ \wedge\ fts|_\lambda \sim lts$.

□