Università di Pisa
Dipartimento di Informatica

Dottorato di Ricerca in Informatica

Ph.D. Thesis: TD-08/03

Non-Functional Aspects of Wide Area Network Programming

Emilio Tuosto

May 2003

Addr: Via F. Buonarroti 2, 56127 Pisa – Italy
Tel: +39-050-2212799 — Fax: +39-050-2212726 — E-mail: etuosto@di.unipi.it
Web page: http://www.di.unipi.it/~etuosto

Thesis Supervisor:
Prof. Gianluigi Ferrari

Abstract

Wide-Area Network (WAN) applications have become one of the most popular applications in current distributed computing. Internet and the World Wide Web are now the primary environment for designing, developing and distributing applications. This scenario imposes different programming metaphors with respect to traditional applications.

Theoretical models for formally reasoning on WAN applications must consider many crucial aspects and their mutual relationships, e.g. mobility, network awareness, security, service level agreement, etc.

This dissertation attempts to formally define declarative approaches for dealing with various facets of actual WAN programming and verification issues.

We propose a declarative approach based on hypergraphs that provide foundational framework for "declaring" components' behaviours of a distributed system. It is exercised with two well-known models for distributed computations as Ambient and KLAIM. Moreover, we extend KLAIM with constructs for specifying, at application level, network connections and related Quality of Service (QoS) requirements. It is also shown how a suitable translation of our KLAIM extension into hypergraphs can be exploited for detecting and reserving optimal routing path with respect to the QoS constraints imposed by applications.

We also introduce a process calculus and a logic that specifies a formal framework for declaring security protocols and properties. By means of symbolic verification technique, the framework can be exploited as a verification environment for model checking security protocols. The declarative flavour of the analysis and the ability of dealing with multi-session verification is the major advantage of the proposed approach.

Finally, we describe a verification environment based on a semantic minimization algorithm for π-calculus agents presented in co-algebraic setting. The final part of this dissertation introduces and discusses an implementation of the algorithm together with other verification facilities of the framework. A proofs that shows the correctness of the implementation with respect to the co-algebraic specification is given.

To my parents, my sister and my brothers for being as they are,
To Eleonora for her incredible patience,
To my artificial family for...

Run, rabbit run
Dig that hole, forget the sun
And when at last the work is done
Don't sit down it's time to dig another one

Breathe (Roger Waters)
The Dark Side of the Moon - March 24th 1973

Acknowledgements

Without any doubt, Prof. Gianluigi Ferrari deserves my gratitude for having followed me during my PhD thesis; Gianluigi has not only been my advisor but a friend too. Giangi, it has been a pleasure and an honour for me to work with you.

During this years in Pisa, I have had the great privilege to work with Prof. Ugo Montanari. Ugo taught me an incredible amount of things and, by observing his approach to research, I also received a lot of "ethical" lessons.

I had the opportunity of knowing and cooperating with Andrea Bracciali, probably the most obstinate person in the department (after me, of course). Andrea it has been a real pleasure to work with you, even during all the dozy nights spent talking, discussing and tex-ing with you.

I am in debt with Roberto Raggi (with whom I implemented Mihda) because he made me remember that programming can suggests new ideas for theory and, anyway, it is very funny. Roberto, be more convinced with respect to your choices and...take it easy.

I want also thank Prof. Rocco De Nicola and Rosario Pugliese for again giving me the possibility of working with them. I am grateful to Rocco and Rosario since the time I worked on my master thesis. They, together with Giangi, introduced me to KLAIM, π-calculus and, more generally, to research. Probably, the first lesson about research was from them: Indeed I learnt that, despite all the issues, research gives a lot of satisfaction.

Probably the "Dipartimento di Informatica di Pisa" is one of the most exiting places for research. Here people that work on very interesting aspects of computer science gave me the privilege of their suggestions and their time. I am grateful to Roberto Bruni, Paolo Baldan (even if Paolo is no longer in Pisa, I consider him as he still were here), Andrea Corradini, Fabio Gadducci, Dan Hirsch and Marzia Buscemi: Thanks to all of you (both for discussions, meals, wines and beers). A special thank to Marzia and Dan for not having asked for a higher temperature in our office during this summer.

I cannot forget to thank Prof. Andrea Maggiolo-Schettini, the coordinator of the PhD course in Pisa. I owe him a lot for having solved many "logistic" issues regarding the timing of my thesis defense.

Mostly, people at the department work seriously. Among them I want particularly to thank Rebecca Micheletti, one of our system administrator for being always gentle: She often went beyond her normal duty in the desperate attempt of filling the holes of my ignorance and my incapacity in managing configurations.

I ringraziamenti sono forse la cosa più prona alle omissioni in una tesi di dottorato. Le persone che non vivono il nostro ambiente faticano ad immaginare la bellezza di lavorare in un dipartimento come il nostro. I miei colleghi (spesso amici) del dipartimento sono in grado di rendere il tempo trascorso al lavoro piacevole (anche senza l'uso di sostanze che possano alterare la percezione del mondo). Sono

sicuro di non essere in grado di fare un elenco esaustivo di tutte queste persone, e dunque non ci proverò... ma grazie a tutti.

I miei genitori non potranno mai essere ricompensati abbastanza per quanto hanno fatto (e ancora fanno) per me e, poichè non sono bravo con parole, cercherò di dimostrare il mio riconoscimento con atti che diano loro le soddisfazioni che meritano. Anna, Carmine ed Enrico mi hanno sempre fatto sentire il loro affetto con forza che si sarebbe dovuta smorzare, considerato il mio carattere. Grazie fratellini, e complimenti per la 'capatosta'.

Da quando ho deciso di intraprendere questa strada la mia famiglia allargata mi ha sempre sostenuto sia moralmente che materialmente in mille discreti modi. Forse non sarà possibile, ma un giorno mi piacerebbe ricambiare almeno un pò quanto hanno fatto per me.

Per tutto quello che mi hanno insegnato sulla vita (nel modo più naturale possibile), un grazie particolare ai miei secondi genitori, nonna Palmina, e nonno Carmine che ci ha lasciato prima che potessi fargli leggere queste righe.

Ho lasciato per ultima Ele. So che non vuoi comparire...ma, infinite grazie per aver condiviso, e spesso alleviato, con me lo stress procurato da questa tesi.

Contents

List of Figures

List of Tables

A Olinda, chi ci va con una lente e cerca con attenzione può trovare da qualche parte un punto non più grande d'una capocchia di spillo che a guardarlo dentro un pò ingrandito ci si vede dentro i tetti, le antenne, i lucernari, i giardini, le vasche, gli striscioni attraverso le vie, i chioschi nelle piazze, il campo per le corse dei cavalli.
In Olinda, if you go out with a magnifying glass and hunt carefully, you may find somewhere a point no bigger than the head of a pin which, if you look at it slightly enlarged, reveals within itself the roofs, the antennas, the skylights, the gardens, the pools, the streamers across the streets, the kiosks in the squares, the horse-racing track.
"Le citt invisibili", Italo Calvino (Hidden cities, Italo Calvino).

Chapter 1

Introduction

Wide-Area Network (WAN) applications have become one of the most important applications in current distributed computing. Indeed, Internet and the World Wide Web are now the primary environment for designing, developing and distributing applications. The problem of supporting WAN computing from specification to architectural design and implementation is at the front-line of the research in the field of Software Engineering since the traditional approaches (e.g. client-server architecture) are no longer sufficient to meet the new demands. New paradigms for WAN computing can be put side by side to previous models in order to achieve the new programming challanges of WAN applications.

One of the most important characteristic of WAN is that control is not centralized. Indeed, WAN have no central point of control and are formed by a number of *administrative domains*. Applications access resources allocated on possibly remote administrative domains and can, at most, control resources allocated on their own administrative domain. One evolutionary programming techniques adopted for deploying WAN applications relies on *web-services* (WS) that can be described as self-contained components which inter-operate with each other by supporting web-based access protocols [102]. Web services may adapt themselves to match the particular capabilities of a variety of devices ranging from traditional PCs, to Personal Digital Assistants and Mobile Phones having intermittent connectivity to the network. In other words, WS can be naturally exploited to deal with highly decentralized and dynamically reconfigurable (e.g. WS are assumed to be *pluggable* to other services to achieve the required functionalities). Moreover, networked heterogeneous applications are becoming the primary part of modern software environments nowadays. These applications (e.g. as web-browsers for cellular phones) require stationary servers and mobile devices to cooperate and exchange services while the application is running.

In this scenario, the dynamic integration of network services is a crucial aspect. Indeed, the services offered by a component are described using an interface description language [47]. Recent developments have focused on extending these languages so as to allow the specification of the interaction protocols of components. Service

integration is obtained by means of composition languages (also called coordination languages) that describe how components are glued together. Furthermore, other issues are relevant when constraints over communications are necessary. For instance, e-conferencing, video on demand or real video applications require that some *quality of services* (in the following, QoS's) are set by applications in order to guarantee an adequate result.

The scenario that we have depicted above induces to consider formal reasoning on WAN applications as a problematic task. Usually, components are stand-alone "black-boxes" that encapsulate services and have standard interfaces. A component reflects service behavioural features through suitable interaction protocols and must be deployed with no assumption on the possible interactions that will be instantiated at execution time. Hence, it is very difficult to state and certify properties of WAN applications. Essentially the reasons of these difficulties can be found in the fact that components are developed by different programmers and along different periods. Moreover, allocation of services and resources can dynamically change at a fast rate and applications are not necessarily aware of those changes. Finally, WAN applications are executed in an unknown environment which also make security issues become primary concerns. Authentication, integrity and secrecy are properties that WAN applications *must* guarantee, but, depending on the application, other properties might be important: For instance we can cite non-repudiation properties in e-commerce applications.

The above considerations drive us to the conclusion that it is very difficult to *virtualize* the execution environment of WAN application even at the very early stages of software development. In particular, applications must take account of resource and control distribution, administrative authorities, coordination and communication characteristics, etc. All those *non-functional* aspects must be considered when a WAN application is designed. Hence, WAN programming requires new innovative methods and techniques to model, specify, design and certify properties.

WAN applications have many peculiarities that distinguish them from traditional applications and that require different programming metaphors, primitives and tools to fit the various facets of computations of WAN applications. A non-exhaustive list of concepts that are strictly related to WAN applications is given by

- interoperability,
- mobility,
- network awareness,
- coordination,
- security,
- service level agreement,

All those items will be later discussed in more detail. The main contributions of this dissertation are all related to the specification, modeling and verification of computations of WAN applications.

- We introduce a graphical model based on *synchronized hyperedge* replacement that has various benefits:
  - The calculus permits to define WAN application in a declarative fashion and in such a way that different conceptual components of the applications are defined independently;
  - The different stages of WAN applications development can be described in a uniform way within the graphical framework. In particular graphical calculus is exploited to refine high level UML specification to more concrete and formal ones.
  - The graphical calculus is also used as intermediate language for mapping foundational models (e.g. the Ambient calculus [39] and KLAIM [19, 57]). The translation of Ambient calculus accounts for a simple interactive semantics for the calculus and the definition of new "Ambient-like" primitives. The translation of KLAIM permits programming attributes related to QoS attributes at application level.
- We introduce a calculus for the formalization of security protocol together with a suitable logic. They allow one to state protocols and their properties in a declarative fashion such that protocols are amenable of being verified in a framework able to consider multiple sessions in a uniform and automatic way.
- We have implemented a verification environment, called Mihda, which allows reasoning on the behavioural properties of WAN specifications. A main property is the direct correspondence between the semantic structures and the implementation structures that facilitates the proof of correctness of the implementation.
  - The algorithm has been specified co-algebraically [73] and the implementation is proved to be correct with respect to the specification. The proof remarks the correspondence between the specification and the implementation.
  - Mihda has been equipped with a web-interface that also permits the integration with different tools. In this way a web-based verification environment is built.

1.1 WAN Features: an Overview

The problem of supporting WAN computing from specification to architectural design and implementation is at the front-line of the research in the field of Software

Engineering since the traditional approaches (e.g. client-server architecture) are no longer sufficient to meet the new demands. In this section we outline our perspective on the current status of the research on WAN computing by identifying the basic concepts and the proper abstractions which are useful in specifying, designing, certifying and implementing WAN applications.

Hereafter, we use the adjective 'distributed' as referred to wide are networks, hence a 'distributed system' is a wide area network and 'distributed programming' refers to programming WAN applications.

Interoperability A key concept in the definition of modern WAN applications is the *interoperability* concept. Applications should be executed on architectures having different hardwares or operating systems. From a programming point of view, interoperability help developers to *write once and run everywhere* their applications This aspects have been already addressed by SUN-MICROSYSTEMS with the definition of Java language [9]. Java face with the problem of mitigating the differences between execution platform. Java tackles the lacking of platform homogeneity by providing an intermediate language, the *byte code*, in which source Java programs are translated. Each platform is then provided with a specific *virtual machine*, i.e. the Java *V*irtual *M*aching (JVM), that essentially implements an interpreter of the byte code for the target machine.

Basically, Java proposes a unique *front-end* for application programmers and multiple *back-end*'s as execution environments.

Recently, MICROSOFT CORPORATION has proposed .NET that further stresses the concept of interoperability. The .NET proposal [124] aims at achieving *language* and platform independence. The idea is the definition of a *Common Runtime Language* (CLR) [91] that provides a unique intermediate language for multiple front-end's. This allows a generalization of the Java programming environment to different programming languages. In this way it is possible to re-compile and integrate applications that had been written in different languages.

Mobility The concept of *mobility* provides a suitable abstraction to design and implement both foundational calculi and languages for WAN programming. Depending on the level of abstraction, different forms of mobility can be considered. We can identify three kinds of mobility:

1. data mobility,
2. link mobility,
3. process/device mobility

Data mobility is the elementary condition for providing communication in a distributed setting.

Link mobility has been extensively studied and seems to be the simplest form of mobility that also provides powerful theoretic tools to study and analyze distributed systems. The π-calculus [162, 129, 130] is probably the most known example of calculus that can model link mobility. The idea is that processes are linked and, during their interactions, they can exchange links, creating new connections and possibility of communications. The most interesting feature of the calculus can be recognized in the possibility of declaring links that are "local" to a process and that can also be exported to other processes. Namely it is possible to *extrude* the scope of a local link. One of the main advantages of the calculus is that it has a robust mathematical theory that allows one to formally reason about systems. In particular, π-calculus systems are considered as *reactive*: they are plugged and executed into an environment that can interact with them by means of *stimuli* to which systems react. We remark that, despite of its conceptual simplicity, link mobility can model many phenomena that arise in WAN programming. For instance, in [161] it has been proved that higher-order calculi (i.e. calculi with process mobility) can be suitably represented in π-calculus. Moreover, slight variations on π-calculus, allow to set the theoretical and practical framework for studying and verifying security protocols [2, 22, 23, 28, 26]; these proposals have great benefits from scope extrusion that is a very appropriate linguistic construct for expressing creation of *nonces*[1] or sharing secrets.

At the higher level of abstraction, a main breakthrough is that WAN applications may exchange active units of behaviour and not just raw data. The usefulness of mobility emerges when developing both applications for nomadic devices with intermittent access to the network (*physical mobility*), and network services having different access policies (*logical mobility*). Process mobility has given rise to new design patterns [52] other than the traditional client-server paradigm:

- *Remote Evaluation*: the code is sent for execution to a remote host;
- *Code On-Demand*: the code is downloaded from a remote host to be executed locally;
- *Mobile Agents*: processes can suspend their execution and migrate to new hosts, where they can resume their execution.

Among these design paradigms, Code On-Demand is probably the most widely used (e.g. Java Applets). The paradigm of mobile agent is, on the other hand, the most challenging since:

- in order to run an agent needs an *execution environment*, i.e. a server that supplies resources for execution;

[1] A *nonce* is random sequence of bits that is normally used in cryptographic protocol "to mark" protocol runs. It is supposed to be unguessable and different from any other datum.

- an agent is *autonomous*: it executes independently of the user who created it (*purpose driven*);
- an agent is able to detect changes in its operational environment and to act according to these changes (*reactivity* and *adaptivity*).

Another interesting feature of mobile agents is the possibility of executing *disconnected operations* [145]: an agent may be remotely executed even if the user (its owner) is not connected; if this is the case, the agent may decide to "sleep" and then periodically try to reestablish the connection with its owner. Conversely, the user, when reconnected, may try to *retract* the agent back home (i.e. instruct the remote agent to come back to its home site).

In addition to this scenario, *ad-hoc networks* [46, 51] allow connection of nomadic devices without needing a fixed network structure. However, ad-hoc networks are at a very early stage of definition and are still not precisely characterized, hence a model for them is still missing. Finally, the shift from client-server to *peer-to-peer* architectures (e.g. Napster, Gnutella) has introduced a new pattern for the Internet interaction where information is shared among distributed components and change dynamically.

Clearly, a formal characterization of the key concepts involved in the development of mobile applications (e.g. QoS, adaptability, resource discovery and usages) is a major concern from a software engineering perspective.

Programming languages and systems provide basic facilities for mobility. A well known example is provided by the Java programming language. Another interesting example is provided by Oracle [142] which supports access to a database from a mobile device by exploiting a mobile agent paradigm.

At a foundational level, several process calculi have been developed to gain a more precise understanding of distribution and mobility. We mention the Distributed Join-calculus [82], KLAIM [57], the Distributed π-calculus [155], the Ambient calculus [39], the Seal calculus [169], and Nomadic Pict [174]. Other foundational models adopt a logical style toward the analysis of mobility. *MobileUnity* [121] and *MobAdtl* [78] are logics specifically designed to specify and reason about mobile systems exploiting a Unity-like proof system. Finally, spatial logic [32, 33, 34] allows one to specify properties on both the spatial dimension and the temporal dimension of WAN applications.

Network Awareness Current software technologies emphasize the notion of WEB SERVICE as a key idiom to control the design and the development of applications. Conceptually, WEB SERVICEs are stand-alone components that reside over the nodes of the network. Each WEB SERVICE has an interface which is network accessible through standard network protocols and describes the interaction capabilities of the service (e.g., the message format). WAN applications are developed by combining and integrating together WEB SERVICEs, which do not have pre-existing knowledge of how they will interact with each other.

The exploitation of components in a WAN setting raises a number of issues. First, given the heterogeneity of the network environment components should be highly portable: components could be used anywhere but require some services to behave properly (i.e. services are used to adapt components to a variety of infrastructures). Second, security should be ensured: components downloaded from different authorities have different security requirements, and they should be executed within different run-time environments. Third, dynamic adaptability should be ensured: WAN applications are highly dynamic and can reconfigure their structure and their components at run-time to respond to dynamic changes of the network environment.

Summing up, a WAN application does not appear as a single integrated computer facility to its users as it is the case of traditional distributed applications. For instance, users of traditional distributed applications can invoke a service regardless of whether the service is local, remote or under the control of a different network authority. Instead, in the WAN setting the *awareness* of network information is crucial for choosing the best services that match user's requirements. For instance, users can react to phenomena like network congestion by binding their network devices to different available resources. Similarly, network awareness is exploited by WAN application designers to control resource usages and resource accesses in order to ensure and maintain certain security levels.

At a foundational level to reflect the idea of network awareness most models exhibit explicit localities, e.g. Ambients [39], KLAIM [57], and MobileUnity [157] to cite a few. Roughly speaking, locations fully identify the network environment of a component. The aforementioned approaches have improved the formal understanding of the complex mechanisms underlying network awareness. For instance, the problem of modeling resource access control of highly distributed and autonomous components has been faced by exploiting suitable notions of type [59, 93, 95, 30, 42].

Coordination WAN applications are highly decentralized and dynamically reconfigurable. Hence, they should be easily scalable in order to manage addition/removal of services, subnetworks and users without requiring to be reconfigured. Coordination is a key concept for modeling and designing WAN applications. Coordination principles separate the computational components from composition modules called *coordinators* which glue components together. Therefore, coordinators offer the basic mechanism to adapt components to the network environment changes, to discover resources, to synchronize activities, and so on. For instance, coordinators are in charge of supporting and monitoring the execution of dynamically loaded modules. Moreover, coordinators are able to observe evolutions, and therefore they may react to an action by modifying themselves. Finally, coordination policies must be programmable to meet the evolving composition demands and to accommodate the design and the implementation of open systems. Two recent examples of coordination middlewares for WAN programming are represented by Jini [10] and .NET Orchestration [124], proposed by Sun and Microsoft, respectively The separation of

concerns between computation and coordination is also at the basis of the research on software architectures [166, 6]

Many approaches to coordination are based on the Linda model [86, 87, 40, 41] which provides the structure of *tuple space* as mechanism to represent the environment of applications. Experimental programming languages and middlewares have been designed following this metaphor [57, 17, 150, 18]. Some preliminary results on defining a discipline for orchestrating WEB SERVICEs is outlined in [8]. The approach is based on the idea of separating WEB SERVICE providers from *contracts* mechanisms (also known under the name of *connectors*), which regulate WEB SERVICE coordination.

The activities in the field of coordination languages and models have improved the formal understanding of dynamically adaptable mobile components. However, the right level of abstraction coordination and the definition of suitable constructs to program crucial policies such as adaptation, loading and security requires further research.

Security Nowadays network computing has to face with distribution of information and computations over local or wide area networks. Accesses to distributed resources must be regulated in order to guarantee that applications cannot read or modify information, unless explicitly authorized. One of the most relevant peculiarities of WAN architectures is the fact that no central authority can impose policies on the access modality to resources that hold for the whole network. Therefore, it is necessary to take care of possible malicious applications that may gain access to resources supposed to be not accessible for them and steal or modify sensible information. Moreover, security is complicated by the impossibility of controlling connections between the sites of the network. Those considerations implies that system security must be faced at different levels: From access control policies to network communication protocols. These considerations have a great impact on the models for WAN programming. All those peculiarities must be considered. As a representative example, it will suffice to consider the amount of efforts that designers for distributed languages spend in order to consider all those characteristics in the very early stages of the definition of the languages primitives. The programmer should be provided with the linguistic mechanisms that permits him/her to define applications that protects their computations and resources from malicious or even "benign" non desired accesses or interactions. In fact, even under the assumption that no malicious participant can take part to computations, there is a very large number of possible connections among services that are not under the control of a single programmer, but rather, they are achieved by interactions of different applications. This implies that WAN programming has to face with a high degree of non-determinism arising from the way in which applications will be connected at execution time. Therefore, unforeseen connections/communications can drive an application to non-desired computations that might be limited with

appropriate linguistic tools that allow applications to control network connectivity and resource accesses. A paradigmatic example is provided by the Java programming language through the SOCKET and the SECURITY APIs [114]. Similarly, the Microsoft .NET [149] architecture supplies a programming technology embodying general facilities for handling heterogeneity. As far as security is concerned, *cryptographic* techniques have been exploited to solve several problems related to security of data communications (authentication, secrecy and integrity). Finally, *firewalls* are barriers that administrative domains build to disable the access to some critical services.

Another implication of security concerns is the robustness of the protocols exploited to perform network interactions. Most of them are *cryptographic* protocols and try to achieve "safe" communications over an non-trusted *medium*. In general this is not enough to avoid undesired interferences. Indeed it is possible to cheat remote partners of a communication even if the cryptographic protocol is supposed to be "perfect", namely if encrypted messages can be understood only by the owner of the appropriate keys and cryptograms can be generated only if the encryption key is owned. In order to limit this kind of attacks/errors many techniques have been developed. Several approaches assume that applications operate in a hostile environment that has "complete" control over exchanged messages (under the constraint of the perfect encryption assumption). More precisely, the environment can modify the sequence of messages, or can forbid communications, forge messages that "resemble" generated by regular applications, and so on.

Finally, an important issue is the analysis and certification of properties of communication protocols. In particular, it should be clearly stated which kind of properties a network communication protocol aims at guaranteeing and under which assumptions this goal is achieved. Therefore, it seems very important to formally analyze protocols adopting formal methods that can provide the necessary confidence in protocol correctness. Many formal frameworks have been defined and implemented for analyzing and verifying security protocols. We cite here few of them [79, 163, 80, 115, 164, 69, 68, 2, 117, 160, 22] and refer to Section 10.5 for more detailed comments.

Service level agreement Network awareness is thus the distinguished and novel issue of WAN applications and refers to the explicit ability of dealing with the unpredictable Quality of Service properties of the network environment. Here with Quality of Service we mean properties such as security, performance, bandwidth, transaction support, cost of service, etc. In general, the perceived QoS of applications is no longer given by the performance of the servers but rather by the availability of certain resources, by the security level provided, by the flow of network traffic, and so on. However, current technologies provide solutions for some aspects related to process/device mobility under very strong simplifying hypothesis. In general, the proposed abstractions have limited expressiveness in order to guarantee safe com-

putations of dynamic and evolutionary WAN applications. Moreover, most research on QoS is *system oriented* by focusing on properties of the lower layers of the Internet protocol stack (e.g. the next version of the Internet Protocol IPv6). Hence, proposed technologies do not allow applications having an explicit and direct control over QoS properties.

The growing demands on security have led to the development of formal models that allow specification and verification of cryptographic protocols. Indeed, the challenge is to formally understand which are the features of an integrated security model for WAN applications. Moreover, the application security policy cannot make any decision using knowledge of the entire current state of the application. Any realistic approach will have to identify which portion of the state of the WAN application is potentially relevant and may affect or be affected by the security policy decisions. Interestingly, the notion of QoS briefly outlined above, may help to investigate the proper trade-offs between expressiveness and security concerns. However, a foundational model dealing with all these facets of network awareness is still missing.

We will concentrate on the aforementioned aspects of WAN computing. We think that they are strictly related. For instance, interoperability, dynamic evolution, code mobility and security are all related and of great relevance for WAN applications. It suffices to think that dynamic linking of remote library requires run-time verification of "untrusted" code and it would be meaningless to imagine infrastructures for interoperant applications that do not consider the relations among those issues [91, 114].

The list of topics discussed above is not exhaustive, it suffice to think that *long term transactions* have a great importance when considered in a WAN scenario and main research is ongoing on those topics.

1.2 Process Calculi

Process calculi have been considered as a formal framework for specifying concurrent or distributed systems and their behaviour [98, 129, 130]. In general, they provide formal operational semantics that give "executable" specifications of various constructs related to systems interactions, resource distribution and accesses, mobility and many other aspects of distributed systems.

As their name suggests, process calculi usually consider systems as obtained by composing processes in an algebraic fashion. They are equipped with a *labelled transition systems* (LTS, for short) that is a set of inference rules which specifies the operational semantics of the calculus. LTS semantics can be defined exploiting the syntactic structure of the terms representing processes [152]. For instance, considering CCS [126], a process may have the form $\alpha.p$ where α is an action. Term $\alpha.p$ must be intended as a process waiting for a synchronizing action with another process offering the complementary action $\bar{\alpha}$. CCS synchronization can take place in a

system built out of two parallel process, e.g. $p \mid q$ that can perform complementary actions. This behaviour is formally expressed in the following inference rule:

$$\frac{p \xrightarrow{\alpha} p' \qquad q \xrightarrow{\bar{\alpha}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'} \tag{1.1}$$

where τ is the *silent* label that represents synchronization. Rule (1.1) shows how the behaviour of the system $p \mid q$ is determined from the behaviours of its subsystems p and q.

A different approach to the semantics of process calculi is the definition of a *reduction relation*. This approach dates back to early nineties when the use of the *Chemical Abstract Machine* (CHAM) was introduced in [16]. This kind of relations specifies how a term can be rewritten because of an *internal reaction*. For instance, a reduction rule that describes an interaction between two CCS processes can be stated as:

$$(\alpha.p + p') \mid (\bar{\alpha}.q + q') \;\rightarrow\; p \mid q \tag{1.2}$$

In some sense, reduction semantics tries to mitigate the rigidity imposed by the syntax. Even though reduction semantics is a familiar concept coming from term-rewriting systems, it is not straightforwardly applicable to process calculi. A main difficulty is that *redexes*, namely the sub-terms of a term that "react", can not be in the form required by rules like (1.2) this imposes to define a structural congruence relation that accounts for transforming the terms in order to narrow reacting sub-terms.

Observation 1.2.1 *Reduction and labelled transition semantics have complementary advantages at the cost of a complementary drawbacks. Reduction semantics is simpler and intuitive than LTS semantics, in general, it is however more difficult to define labelled transition systems instead of reduction relations. One of the main advantages of labelled transition systems with respect to reduction semantics is that behaviours are related to the syntax, therefore the structure of the proofs about behaviours follows the term representing the system. Moreover, reduction relation does not give any account of the behaviour of a system with respect to its environment, while LTS's describe observational semantics.*

Observational equivalences, disregarding the structure of states in a LTS, relates systems that offer "the same" behaviour to the surrounding environment. Many different equivalences have been given, for instance under *testing equivalence* [60] two systems are equivalent if they "pass" the same class of tests, where a test is an "observer", expressed as a process that executed in parallel with the system can or not fire a "success" transition represented by a distinguished label.

A finer equivalence is *bisimulation* that can be defined for process calculi equipped with interactive semantics. If we think of LTS as an automaton where the states are the terms of the process calculi and the transitions are the transitions

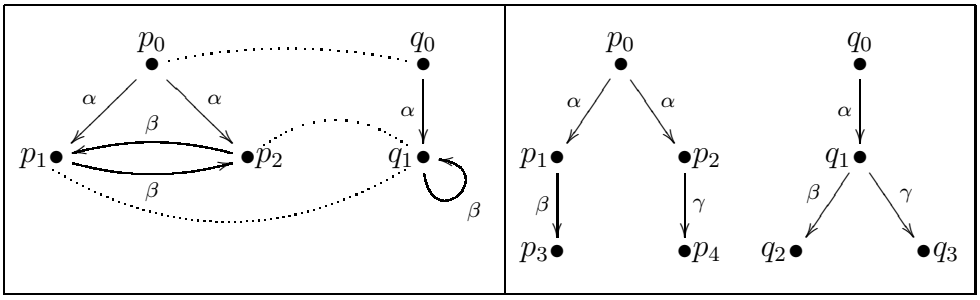


Figure 1.1: Bisimulation

that can be derived from the inference rules of the process calculus, then we can lift bisimulation relation on processes to bisimulation on automata. Process bisimilarity is an important conceptual instrument both for theoretic investigation and for practical issues.

From a theoretic point of view, it provides a natural behavioural equivalence for reactive systems. Moreover, in some cases, it is interesting to study under which conditions concerning to LTS inference rules of a process calculus the resulting bisimulation is a congruence with respect to the operators of the calculus, in order to achieve compositionality.

On the practical perspective, if we think of process calculi as specification and implementation languages, we can use bisimilarity to check whether a system specification matches its implementation. We can also apply minimization algorithms that preserves bisimilarity and then use the minimal realization of the automata for model checking properties of the system. Roughly, two automata are bisimilar if there is a binary symmetric relation between their states such that their initial states are in that relation and, whenever two states are related, each transition of one of the states has a corresponding transition from the other one and the corresponding target states remain bisimilar. The left part of Figure 1.1 represents two bisimilar automata, where the related states are connected by dotted lines, while on the right part two non-bisimilar automata are depicted. Let us observe that the non-bisimilar automata accept the same traces (i.e. $\{\alpha\beta, \alpha\gamma\}$) implying that bisimilarity differs from the classical equivalence of automata that consider equivalent two automata if, and only if, they recognize the same language.

1.3 Hypergraphs

Foundational researches on global computing aim at describing, by modeling and analyzing the complex interactions taking place in inter-network computations encompassing several physical networks, multiple administration domains and a variety of possible users. Several models have been proposed to tackle the new computational phenomena.

Most of the proposed models for distributed computing mainly focus on the

spatial structure of systems. To reflect the idea of administration domains, these models exhibit explicit localities, which help to model distributed computations and the discovery of network resources and services. As a paradigmatic example we can cite KLAIM [57] or Ambient calculus [39]. These features distinguish the models of global computing from traditional models (and paradigms) for distributed programming (e.g. CORBA [140]), the motto being *network awareness*: localities are under programmer's control.

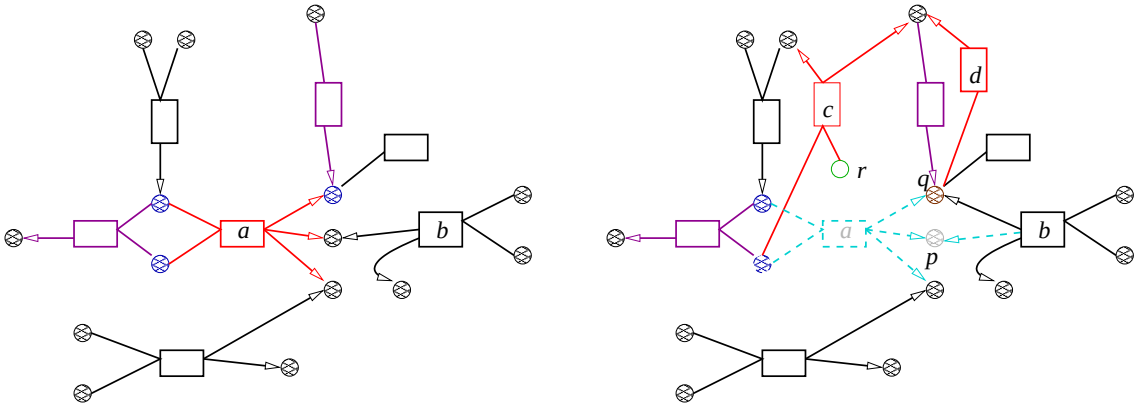
However, network awareness is only one relevant tile of the mosaic of global computing. Another important aspect concerns the *temporal structure* of computations. The run-time environment typically interleaves computational activities with structuring and managing activities. Temporal structure takes care of describing how rearrangements and security checks take place along computations. A proper understanding of both spatial and temporal structures is clearly needed to allow formal verification.

Graph-based techniques can be usefully adopted for modeling inter-networking systems. Indeed, hyperedges can be used to represent components and nodes to model the network environment of components. Edges sharing a node means that the corresponding components may interact by exploiting network communication infrastructure. Structured versions of graphs (typed graphs, term graphs, hierarchical graphs) can precisely model complex inter-network configurations [96, 97] and access control policies [107, 108, 109].

Graph synchronization adds to network awareness the ability of dealing with the temporal dimension of computations. Graphs synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that local rewritings depends on the outcome of a (possibly global) constraint satisfaction algorithm. Nodes can be exchanged during synchronizations, hence constraint solving must encompass unification in order to fuse node.

We propose a new edge rewriting mechanism that allows interconnection modification. The idea is that an edge can be rewritten if the condition it imposes on its synchronization nodes are matched with all other edges connected to such nodes. Figure 1.3 aims at giving a graphical intuition of edge replacement. The edge a in Figure 1.3(a) is replaced by a new graph made of edges c and d connected as shown in Figure 1.3(b), where the initial situation is represented by the dashed part of the graph that disappears after the synchronization. Note that the tentacle of the edge b is attached to a different node because nodes p and q have been fused. This amounts to *mobility* of components, that dynamically can change their connections. Moreover, notice that, after the transition, node r is created, whereas the rest of the graph is not changed. The edge substitution is triggered by the fact that, all the constraints imposed by the edge at its interface nodes are matched by all the other connected components.

One may wonder if this approach is too abstract and general and it does not capture the intrinsic limitations of inter-networking computations. We feel that on the one side the generality of the approach can be tamed and adapted to the needs



(a) (b)

Figure 1.2: Hyperedge replacement

of the various layers of applications, more powerful primitives being made available to upper layers, like *business to business* (B2B) or *computer supported collaborative work* (CSCW). On the other side, some important network technologies actually require the solution of global constraints, like modifying local router tables according to the routing update information sent by the adjacent routers.

Graph rewriting based on edge replacement and synchronization was introduced in [43, 61] and related to distributed constraint satisfaction problems in [136]. The version with mobility, which employs a notation based on logical sequents and inference rules, was introduced recently in [96] and extended in [97] to encode π-calculus. Abstract semantics based on bisimilarity has been discussed in [110].

1.4 Verifying Properties

One of the main advantages of applying formal methods to system design and specification is the possibility of constructing an abstraction of systems and their computations that are, at least at a certain extent, amenable of automatic verification of properties. Many techniques have been studied, implemented and used for verifying systems. Many of them are based on *simulation*, *testing* or *deductive reasoning*. Recently, one of the most successful formal technique for verification of systems finite state systems is *model checking*.

Simulation and testing techniques requires that some experiments are conducted on a prototypical realization of the system (testing) or an abstract model of it (simulation). In both cases the idea is to submit the (model of the) system to some stimuli at its input interface and detect the results at output interface. Even if many errors can be found using these techniques, however, their main limitation is that

not *all* errors can be generally detected because of the number of possible inputs is infinite, or the number of possible input configurations is very large.

Deductive reasoning is probably the most intriguing method for system verification and has significantly influenced software development for instance, the notion of *invariant* is a product of this line of research. Probably, the most important theoretical base for deductive reasoning can be recognized in type theory [119, 120], and the Curry-Howard isomorphism [100, 53] which states that propositions can be interpreted as types and proofs as programs. Indeed, by exploiting the Curry-Howard isomorphism it is possible to state a property ϕ as a type τ_ϕ. If a program which has type τ_ϕ can be built then the program enjoys ϕ. This idea had been applied to obtain several tools like "proof assistants" or proof editors. Proof assistant drive the programmer in writing programs which has the desired type. In fact, in general, this is not a mechanizable task and requires human intervention [62, 55, 139, 148]. This essentially constitutes the main drawback for deductive methods because many expertise is required to interact with proof assistants.

Roughly, model checking describe system behaviours as state transitions and properties that must be checked usually are temporal logic formulae whose models are state transitions representing behaviour of systems that essentially provide the *Kripke* structures of interest. In order to model check a system with respect to a given formula it is necessary to check whether its state transition is a logical model of the formula. Model checking has been extensively used for hardware and software verification and has some advantages over other methods:

- by exploiting very refined data structures (e.g. BDDs), or symbolic techniques, it is possible to obtain very high efficiency;
- usually, model checkers provide counterexamples when a system does not satisfy some property. This gives to designers and implementors information on design choices of implementation errors.
- in general model checking is completely automatic, provided that finiteness of the model is guaranteed.

On the other hand, model checking cannot deal with infinite state systems which is instead possible with deductive techniques. Perhaps a very useful line of research is the integration of deductive and model checking techniques for having the benefit of both the two methods.

1.5 Main Contributions

WAN applications impose new challenges in defining theoretical frameworks, linguistic paradigms and tools for analyzing and verifying distributed systems. Several techniques and programming paradigms have been proposed (e.g. those cited in Section 1.1). However, they suffer the lack of formal methods that can uniformly

be applied to reason on deployed distributed systems. Indeed, in this context a theoretical framework that can encompass all (or, at least, many) of the issues raised from WAN applications' deployment is still lacking.

This dissertation attempts to formally define declarative approaches for dealing with various facets of actual WAN programming and verification issues. We try to contribute to the distributed programming along two principal directions.

The first achievement is related to WAN programming.

1. We propose a declarative approach based on a hypergraphical calculus that extends the calculus proposed in [97] allowing arbitrary fusions on nodes. This approach tries to provide a formal framework that allows programmers to "declare" the components' behaviours of a distributed system.

2. In Chapter 5 it is shown how the graphical calculus can be exploited for encoding the well-known Ambient calculus. Theorems 5.4.1 and 5.4.2 formally give evidence to the fact that Ambient requires multi-party synchronizations and arbitrary node fusions for being faithfully represented with hypergraphs.

3. In Chapter 6 the QLAIM calculus is presented. QLAIM has been obtained as a natural extension of the KLAIM dialect proposed in [18]. The key features of QLAIM are its mechanisms for specifying, at application level, network gateways equipped with QoS requirement.

4. We have also proved that QLAIM can be mapped in the hypergraphs calculus. As a major benefit of the encoding Theorem 6.3.1 and Section 6.4 show that the graphs obtained by the translation can detect and reserve the optimal routing path with respect to the QoS constraints imposed by the application.

A second achievement is related to verification of properties of distributed applications.

1. The decidability result of [48] is exended to the case of asymmetric key cryptography in Section 9.3.

2. We introduce a process calculus and a logic that, starting from the ideas of [130, 2, 22], specifies a formal framework for declaring security protocols and properties.

3. Symbolic verification techniques can be exploited for model checking security properties of cryptographic protocols specified in our framework.

4. A peculiarity of our verification environment with respect to similar proposals, lies in the declarative flavour of the analysis and the ability of dealing with multi-session verification automatically.

Finally, we describe a verification environment based on a semantic minimization algorithm for π-calculus agents presented in co-algebraic setting. The co-algebraic presentation of the algorithm has been given in [73].

1. The final part of this dissertation introduces and discusses an ocaml implementation of the algorithm together with other verification facilities of the framework.

2. A proofs that shows the correctness of the implementation with respect to the co-algebraic specification is given. The implementation can be exploited for checking π-calculus agents bisimilarity.

As a final comment, we would like to suggest a possible interpretation of the results. All the proposed frameworks are very centered on the concept of naming that has pervasively been applied to many concurrent and distributed frameworks for the last decades.

The hypergraphical calculus we propose can be seen as a "declarative" version of the π-calculus. Indeed, many of the π-calculus features have been inherited by our calculus. The innovative aspect is the possibility of imposing constraints on the interfaces of distributed components and to exploit those constraints for expressing the hyperedge replacement mechanism. This aspects fits well with the mechanisms that have been defined by other π-like calculi as Ambient or KLAIM. Moreover, it can also suggests new primitive in the style of those calculi as will be clear in Section 5.5.

Similarly, the proposed calculus and logic for security protocol analysis can be seen as π-calculus mechanisms variants that adapts to the modeling issues related to security protocols and properties.

Finally, the implementation of the minimization algorithm has been designed to be easily extensible to name-passing calculi. Indeed, once the co-algebraic functor of the name passing calculus has been defined[2], by exploiting the polymorphic type system of ocaml, it is easy to reuse the same algorithm for the new calculus.

1.6 Outline of the Thesis

Chapter 2 collects some notational conventions that we have adopted through the dissertation and represents a quick reference for notational concerns. (Notations that are used only in certain point of the thesis are introduced when needed.)

Chapter 3 outlines some technical background that is essential to the understanding of the remaining part of the thesis. In particular

- Section 3.1 reports some necessary definitions on π-calculus which is the underlying model of all the proposed frameworks.

[2]The functor must satisfy some hypothesis that will be given in Section 12.1.

- Section 3.2 reports syntax and semantics of the Ambient calculus.
- Section 3.3 reports syntax and semantics of the KLAIM.
- Section 3.5 informally discusses some issues related to security protocols[3].
- Section 3.4 tries to give account of other approaches that, even though not fully considered in our presentation, must be mentioned in order to relate them to our approach.

The original contributions of the thesis are organized in three different parts, while Part IV resumes some concluding remarks and discusses some possible future directions of our research. Parts I, II and III begin with short abstracts that resume the results. Despite of some redundancy, we hope that this should help reading.

Part I details the hypergraphical calculus that we propose as formal programming model of WAN applications together with QLAIM, a calculus propose for specifying QoS attributes at application level. This part also contains the translations of Ambient and KLAIM into the hypergraphical model. It is divided into four chapters.

- Chapter 4 introduces hypergraphs, their syntax and the semantics based on hyperedge replacement. The final section of the chapter introduces a hyperedge replacement semantics that allows multiple hyperedge replacements to be contemporary applied.
- Chapter 5 details the translation of Ambient into the hypergraph model. It is proved that the translation "preserves" the semantics of the calculus. Last section of the chapter contains the definitions of new Ambient-like primitives that can easily be defined on the graphs obtained by translating Ambient processes.
- Chapter 6 gives some motivations for the need of primitives for specifying QoS requirements at the level of WAN applications. Then a slight variation of KLAIM is introduced in order to obtain QLAIM, a calculus that explicitly allows programmers to express QoS constraints in their applications. Section 6.3 details how QLAIM nets can be mapped into hypergraphs that respects their semantics. Moreover, it is shown how such hypergraphs can be exploited for searching and reserving the optimal routing path, where the optimality criteria are imposed by the QoS requirement imposed at the application level.
- Chapter 7 presents a methodology exploited to use the model of hypergraphs as a formalization of UML specifications. The added value of the translation lies in the possibility of using the obtained hypergraphs as a distributed

[3]We have preferred to postpone the technicalities of this part to PART II because many formalizations have been proposed for security protocols and we think that it is better for the non-expert reader to be first narrowed to the topic by means of an informal discussion.

architecture even though the initial system had not been specified with any distribution purpose.

Part II is devoted to the analysis and verification of security protocols. This part details a process calculus for specifying cryptographic protocols and, on the top of the calculus, a logic for expressing security properties. This part is constituted by three chapters.

- Chapter 8 collects some basic notions of cryptography, protocols and security properties that allows us also to recap some notations and terminology normally adopted in security literature.
- Chapter 9 first illustrates the intruder model of Dolev-Yao that is normally assumed in analysis of cryptographic protocols, then the informal description of the intruder is formalized and exploited in the definition of the cryptographic calculus proposed in this dissertation (called cIP). The final section of the chapter introduces the logic for expressing security protocols and defines its models in terms of the computation traces of the cIP calculus.
- Chapter 10 defines a symbolic semantics of cIP in order to permit finite model checking of properties expressed in the proposed logic. We show how the symbolic semantics can faithfully consider the "concrete" traces of cIP that lead to attack of the protocol (if any).

Part III describes an implementation of a semantic minimization algorithm for π-calculus agents. The algorithm exploits HD-automata for representing transition systems of π-agents. Then HD-automata are minimized by collecting bisimilar states in a single state and by considering only the names that have meaning. The theoretical framework of the algorithm lies on a co-algebraic setting that is reported as a necessary background of Part III. This part is divided in four chapters.

- Chapter 11 reviews notions of algebras and co-algebras. In particular, it is recalled how automata can be suitably described as co-algebras of particular functors.
- Chapter 12 first outlines HD-automata by considering their main advantages in representing labelled transition systems of π-agents, then the co-algebraic functor which allows us to define the minimization algorithm as an iterative procedure that constructs the final element in a suitable co-algebra is described.
- Chapter 13 details Mihda, that is the ocaml implementation of a verification environment based on the minimization algorithm of HD-automata. The main aspects of the implementation are introduced and discussed and it is also shown how they are related to the co-algebraic specification of the algorithm.

- Chapter 14 describes a web interface that has been defined in order to provide a web-based verification environment obtained by interfacing Mihda and the pre-existing history dependent automata laboratory (HAL).

1.7 Origins of the Chapters

Some of the results presented in this dissertation have been submitted for publication, others have been presented in some conferences and some of them have been already published in preliminary form. However, many changes have been made with respect to the published material. Even though the formalisms have usually been refined and the material has been extended in this thesis, the basic ideas behind the results remain the same, hence we give a list of references that point to the paper that describe the initial versions of the works.

- Hypergraphs with synchronized hyperedge replacement and the mapping of the Ambient calculus have been introduced in [76].
- The application of hypergraphs to KLAIM model of distributed programming and application level QoS has been detailed in [56];
- Relationships between hypergraphs, hyperedge replacement and software design has been presented in [77].
- Security issues have been investigated in [27, 28].
- Mihda and the related web verification environment have been described in [75] and in [72].

Chapter 2

Notations

This chapter resumes the main notational conventions that will be adopted through this dissertation. We intend to give the reader the possibility of quickly referring this chapter when doubts on some technicalities may arise. However, we warn that other notations (adopted only in some parts of the thesis) will be introduced when needed.

Sets In the next chapters there will be many definitions where operation on sets involves singletons. In order to avoid clumsy parenthesized expressions we will abbreviate such set-theoretic operations as follows:

$$A \setminus a \stackrel{\text{Not}}{=} A \setminus \{a\}, \qquad A \cup a \stackrel{\text{Not}}{=} A \cup \{a\}, \qquad A \cap a \stackrel{\text{Not}}{=} A \cap \{a\}.$$

Given a set A, $\wp(A)$ denotes the set of all subsets of A. Since we must frequently consider *finite* subset of a given A, we introduce the notation $\wp_{\text{fin}}(A)$ for indicating the set of all finite subset of A.

Functions If we want to explicitly remark that a function $f : A \longrightarrow B$ is an injective or bijective function, we write

$$f : A \stackrel{inj}{\rightarrow} B, \qquad f : A \stackrel{bij}{\rightarrow} B$$

respectively.

Through the dissertation, we will often deal with partial functions, namely functions $f : A \longrightarrow B$ that can possibly be not defined on some elements of A. In this case, we write $f(a) \downarrow$ ($f(a) \uparrow$) to say that f is defined (undefined) on a. Moreover, we define

$$dom(f) = \{a \in A : f(a) \downarrow\} \qquad cod(f) = \{f(a) \in B : a \in dom(A)\}.$$

and write $f : A \multimap B$ to indicate the fact that f is a partial function from A to B.

Substitutions Through many parts of the thesis substitutions will be used. It is worth to fix some syntactical conventions that will permit to have compact representations for substitutions.

Substitutions of names, typically are extensively used and necessary for modeling semantics of name passing calculi. In this case, if $\mathcal{N}$ is the (infinite) set of names, a substitution on $\mathcal{N}$ is function $\sigma : \mathcal{N} \to \mathcal{N}$ such that

$$\{n \in \mathcal{N} : \sigma(n) \neq n\} \text{ is finite,}$$

whereas, for infinitely many names σ behaves as the identity. This definition allows us to represents substitutions with expressions of the form

$$[{}^{n_1}/_{m_1}, ..., {}^{n_h}/_{m_h}] \tag{2.1}$$

that represents the substitution that replaces m_i with n_i for each $i = 1, ..., h$. We will sometimes prefer the more compact $[{}^{n_1,...,n_h}/_{m_1,...,m_h}]$ notation to (2.1).

If t is a term that contains free and bound names, then $t\sigma$ denotes the term obtained by replacing free occurrences of names according to σ. In general, application of a substitution to a term requires α-conversions of bound names to avoid name capture phenomena; in those case, we assume that α-conversions are silently exercised. Finally, as usual, $t\sigma\sigma'$ reads as $(t\sigma)\sigma'$.

In Part II we use substitutions that maps variables into terms. We will adopt the same notations detailed above.

Chapter 3

Background: WAN Programming: Models, Issues

Contents

3.1 The π-calculus

The π-calculus [130] is the best known example of core calculus for mobility. It is centered around the notion of *naming*: mobility is achieved via *name passing*. Channel names can be created, communicated and are subjected to sophisticated scoping rules. The capability of exchanging channel names gives π-calculus the ability of dynamically reconfiguring process acquaintances.

Name passing primitives are simple but expressive; indeed π-calculus can model objects (in the sense of object oriented programming [172]) and higher order communication [161].

In this section we outline the syntax and the early semantics of the calculus and refer the reader to [130, 162, 129] for a detailed presentation of the variegated facets of π-calculus.

3.1.1 Syntax

We assume as given an infinite set of names $\mathcal{N}$ and we let $a, b, \ldots, x, y, \ldots$ to range over $\mathcal{N}$. Agents of π-calculus are built over terms generated by the following productions:

$$\begin{array}{lcl} p & ::= & \mathbf{0} \mid \alpha.p \mid p \mid q \mid p+q \mid (\nu\, y)p \mid [x=y]p \mid A(x_1, ..., x_n) \\ \alpha & ::= & \tau \mid x(y) \mid \overline{x}y \end{array} \tag{3.1}$$

A process can be the void process, a process prefixed with actions, the parallel composition of processes, the non-deterministic alternative between two processes, a process obtained by restricting a name, a process guarded by equality of names or the recursive invocation of an agent. In (3.1) we let A to range over a set of process identifiers and, for each A, we assume that

- there is a unique definition $A(y_1, ..., y_n) \stackrel{\triangle}{=} q$ where the y_i's are all distinct and $\mathrm{fn}(q) \subseteq \{y_1, ..., y_n\}$;
- whenever A is used, its arity is respected;
- if $A(y_1, ..., y_n) \stackrel{\triangle}{=} p$ is the definition of A, each process identifier in p is in the scope of a prefix (guarded recursion).

Actions of π-calculus are

- τ, also called *silent* action, that represent non-observable or internal computation,
- *input action* $x(y)$ representing the reception along channel x of a name to be replaced for y,
- *output action* $\overline{x}y$ represents the output of name y along channel x.

α	$\mathrm{fn}(\alpha)$	$\mathrm{bn}(\alpha)$	$\mathrm{n}(\alpha)$
τ	$\emptyset$	$\emptyset$	$\emptyset$
$x(y)$	$\{x\}$	$\{y\}$	$\{x, y\}$
$\overline{x}y$	$\{x, y\}$	$\emptyset$	$\{x, y\}$

Table 3.1: Free and bound names of π-calculus prefixes

For input and output action we call x the *subject* and y the *object* name, respectively.

The input action and the restriction operator $x(y).\_$ and $(\nu\, y)\,\_$ act as binders for name y with scope the argument process. However, they have different nature: in the first case, y indicates the placeholders where the received name must be placed; in the second case, y is a new, private name. Notions of *free names* of a prefix action α, $\mathrm{fn}(\alpha)$, of *bound names* of α, $\mathrm{bn}(\alpha)$ arise as expected and are reported in Table 3.1. Given a process p, we can define *free names* of p, $\mathrm{fn}(p)$ and *bound names* of α, $\mathrm{bn}(\alpha)$ as done in Table 3.2, where $A(y_1, ..., y_n) \stackrel{\triangle}{=} q$. The set of *names* of p is

p	$\mathrm{fn}(p)$	$\mathrm{bn}(p)$
$\mathbf{0}$	$\emptyset$	$\emptyset$
$\alpha.q$	$(\mathrm{fn}(\alpha) \cup \mathrm{fn}(q)) \setminus \mathrm{bn}(\alpha)$	$\mathrm{bn}(\alpha) \cup \mathrm{bn}(q)$
$q_1 \mid q_2$	$\mathrm{fn}(q_1) \cup \mathrm{fn}(q_2)$	$\mathrm{bn}(q_1) \cup \mathrm{bn}(q_2)$
$q_1 + q_2$	$\mathrm{fn}(q_1) \cup \mathrm{fn}(q_2)$	$\mathrm{bn}(q_1) \cup \mathrm{bn}(q_2)$
$(\nu\, y)q$	$\mathrm{fn}(q) \setminus y$	$\mathrm{bn}(q) \cup y$
$[x = y]q$	$\mathrm{fn}(q) \cup \{x, y\}$	$\mathrm{bn}(q)$
$A(x_1, ..., x_n)$	$\mathrm{fn}(x_1, \cdots, x_n)$	$\emptyset$

Table 3.2: Free and bound names of π-calculus processes

the set $\mathrm{n}(p) = \mathrm{fn}(p) \cup \mathrm{bn}(p)$. We shall write $\mathrm{fn}(p, q)$ in place of $\mathrm{fn}(p) \cup \mathrm{fn}(q)$ (similarly for $\mathrm{bn}(\cdot)$ and $\mathrm{n}(\cdot)$).

We adopt the following usual syntactic conventions: $\alpha.p \mid q$ stands for $(\alpha.p) \mid q$, $(\nu\, x)p \mid q$ for $((\nu\, x)p) \mid q$ and $(\nu\, x_1 \ldots x_m)p$ for $(\nu\, x_1) \ldots (\nu\, x_m)p$. Moreover, trailing occurrences of $\mathbf{0}$ shall usually be omitted.

A *structural congruence* relation, $\equiv$, is defined on π-calculus agents. It is the least congruence relation that satisfies the axioms in Table 3.3. The structural congruence basically provides an equational algebra for manipulating and rearranging processes without affecting their behaviour and simplifying the rules of operational semantics. For instance, process $a(x).\overline{x}b \mid (\nu\, y)\overline{a}y$ is structurally equivalent to $(\nu\, y)(a(x).\overline{x}b \mid \overline{a}y)$ because we can enlarge the scope of the ν operator. Moreover, structural congruence performs some garbage collection of dead processes.

It is worth to add some comments to the syntax of the calculus in order to give an intuition of the meaning of its terms before giving their formal semantics. Process $\mathbf{0}$ stands for a process that has no possibility of interacting with other processes.

(ALPHA)	processes which differ by α-conversion are equivalent			
(PAR)	$\mid$ is associative and commutative, and $\mathbf{0}$ is its identity			
(SUM)	$+$ is associative and commutative and $\mathbf{0}$ is its identity			
(SCOPE)	$p \mid (\nu\, a)\ q$	$\equiv$	$(\nu\, a)(p \mid q)$	if $a \notin \mathrm{fn}(p)$
(RES)	$(\nu\, a)\ (\nu\, b)\ p$	$\equiv$	$(\nu\, b)\ (\nu\, a)\ p$	
(NIL)	$(\nu\, a)\mathbf{0}$	$\equiv$	$\mathbf{0}$	
(MATCH)	$[a = a]\mathbf{0}$	$\equiv$	$\mathbf{0}$	

Table 3.3: π-calculus structural congruence

A process prefixed by an action $\alpha.q$ can evolve after the action α has been fired; if α is the silent action, then the process evolves without synchronizing with other processes, otherwise another process must offer a complementary action[1] for the synchronization: While such action is not offered, the process cannot evolve. Parallel composition of q_1 and q_2, $q_1 \mid q_2$, evolves to $q_1' \mid q_2$ when q_1 evolves to q_1' without interacting with q_2 (and similarly for q_2), or evolves to $q_1' \mid q_2'$ when q_1 and q_2 synchronize. Differently, if q_1 evolves to q_1' then non-deterministic choice $q_1 + q_2$ evolves to q_1' too disregarding the alternative q_2 (and similarly for q_2). As stated above, $(\nu\, y)p$ hides y to the processes outside the scope p; we will see that the scope of the restriction can dynamically change. Matching-guarded process $[x = y]p$ evolves as p only if the condition $x = y$ holds, otherwise it is stuck. Finally, if $A(y_1, ..., y_n) \stackrel{\triangle}{=} q$, then a recursive invocation $A(x_1, ..., x_n)$ is the same as $q[^{x_1,\cdots,x_n}/_{y_1,\cdots,y_n}]$, namely, as the process obtained by substituting each free occurrence of the formal parameter y_i with the actual parameter x_i.

3.1.2 Early semantics of π-calculus

The early semantics of π-calculus was first introduced in [131]. We report here a slightly simplified variation given in [151]. The labels of the labelled transition system for the early semantics of π-calculus are specified by the following productions:

$$\mu \ ::= \ \tau \ \mid \ xy \ \mid \ \overline{x}y \ \mid \ \overline{x}(y)$$

and are respectively called *synchronization*, *free input*, *free output* and *bound output* actions. Table 3.4 reports the definition of *free names*, *bound names* and *names* of a label μ, respectively written as $\mathrm{fn}(\mu)$, $\mathrm{bn}(\mu)$ and $\mathrm{n}(\mu)$. The labelled transition system for the early semantics of π-calculus is specified by the rules in Table 3.5. Let us remark that actions are different from prefixes because the free input and the bound output actions are not prefixes. Indeed input prefixes have the form $x(y)$ while free inputs are xy. The notation should be reminiscent of the fact that input prefixes act

[1] if α is an output on x the complementary actions are input actions on x, or else output actions on x, if α is an input on x.

μ	$\mathrm{fn}(\mu)$	$\mathrm{bn}(\mu)$	$\mathrm{n}(\mu)$
xy	$\{x,y\}$	$\emptyset$	$\{x,y\}$
$\overline{x}y$	$\{x,y\}$	$\emptyset$	$\{x,y\}$
$\overline{x}(y)$	$\{x\}$	$\{y\}$	$\{x,y\}$
τ	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.4: Free and bound names of π-calculus labels

$$[tau] \quad \tau.p \xrightarrow{\tau} q \qquad\qquad [out] \quad \overline{x}y.p \xrightarrow{\overline{x}y} p$$

$$[in] \quad x(z).p \xrightarrow{xy} p[{}^{y}/_{z}] \qquad\qquad [sum] \quad \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'}$$

$$[par] \quad \frac{p \xrightarrow{\mu} p'}{p \mid q \xrightarrow{\mu} p' \mid q} \text{ if } \mathrm{bn}(\mu) \cap \mathrm{fn}(q) = \emptyset \qquad [comm] \quad \frac{p \xrightarrow{xy} p' \qquad q \xrightarrow{\overline{x}y} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$$

$$[match] \quad \frac{p \xrightarrow{\mu} p'}{[x=x]p \xrightarrow{\mu} p'} \text{ if } x \notin \mathrm{bn}(\mu) \qquad [res] \quad \frac{p \xrightarrow{\mu} p'}{(\nu\, x)p \xrightarrow{\mu} (\nu\, x)p'} \text{ if } x \notin \mathrm{n}(\mu)$$

$$[open] \quad \frac{p \xrightarrow{\overline{x}y} p'}{(\nu\, y)p \xrightarrow{\overline{x}(y)} p'} \text{ if } x \neq y \qquad [close] \quad \frac{p \xrightarrow{xy} p' \qquad q \xrightarrow{\overline{x}(y)} q'}{p \mid q \xrightarrow{\tau} (\nu\, y)(p' \mid q')} \text{ if } y \notin \mathrm{fn}(q)$$

$$[rec] \quad \frac{p[{}^{x_1,\cdots,x_n}/_{y_1,\cdots,y_n}] \xrightarrow{\mu} p'}{A(x_1,...,x_n) \xrightarrow{\mu} p'} \text{ if } A(y_1,...,y_n) \triangleq p \qquad [cong] \quad \frac{p \equiv p' \qquad p' \xrightarrow{\mu} q' \qquad q' \equiv q}{p \xrightarrow{\mu} q}$$

Table 3.5: Early semantics of π-calculus

as binders for the object variables, instead the objects in free inputs are the effective received values in input actions. The bound output transitions are the peculiarity of the π-calculus. A bound output transition represent the communication of a name that has previously been restricted and, therefore, it corresponds to the generation of a name new with respect to "the names of the environment". This mechanism is called *name extrusion* and is formalized by the interplay between rule [*open*] and rule [*close*]. Rule [*open*] reads as: if p can perform a free output transition $\overline{x}y$ and continues as p' then $(\nu\, y)p$ can make a bound output transition $\overline{x}(y)$ and continues as p', provided that $x \neq y$. Note that after bound output transition y is no longer restricted. If a synchronization involving a bound output and a free input action takes place, after the transition we restrict again the "newly generated" name y. Side conditions of the rules [*par*], [*open*] and [*close*] are necessary for avoiding name capture of free names. The remaining rules, are basically the formalization of their informal description given at the end of Section 3.1.1.

Observation 3.1.1 *An important aspect concerning early semantics and verification of π-agents must be remarked. The peculiarity of early semantics lies in the rule [in] that instantiates the object name when the input transition is derived. This implies that a process $x(z).p$ can trigger an infinite number of transitions (one for each instantiated name y). If we think of π-agents as nodes in an automaton, this gives rise to infinite branch on any input node. We will discuss this issues in deeper detail in Chapter 12.1.*

Now we present the definition of early bisimulation for π-calculus.

Definition 3.1.1 (Early bisimulation) *A binary relation $\mathcal{R}$ over π-agents is an* early bisimulation *if, whenever $p\mathcal{R}q$ then*

> *for each $p \xrightarrow{\mu} p'$ such that $\mathrm{bn}(\mu) \cap \mathrm{fn}(p, q) = \emptyset$, there is some $q \xrightarrow{\mu} q'$ such that $p'\mathcal{R}q'$.*

Two π-agents are early bisimilar*, written $p \sim q$, whether there is a bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.*

Condition $\mathrm{bn}(\mu) \cap \mathrm{fn}(p, q) = \emptyset$ in Definition 3.1.1 is necessary to guarantee that the name chosen to represent the newly created name in a bound output transition is "fresh" for both agents. The following example should make more clear the need for name freshness in bound output transitions.

Example 3.1.1 *Let us consider the π-agents $p \equiv (\nu\, y)\overline{x}y.\mathbf{0}$. It is easy to see that the transition*

$$(\nu\, y)\overline{x}y.\mathbf{0} \xrightarrow{\overline{x}(y)} \mathbf{0}$$

can be inferred by rules [out] and [open] in Table 3.5. Intuitively, p should not be distinguished from

$$q \equiv (\nu\, y)\overline{x}y + (\nu\, z)\overline{z}w$$

$$[in^l] \quad x(y).p \xrightarrow{x(y)} p \qquad [comm^l] \quad \frac{p \xrightarrow{x(y)} p' \qquad q \xrightarrow{\overline{x}z} q'}{p \mid q \xrightarrow{\mu} p'[^z/_y] \mid q'}$$

$$[close^l] \quad \frac{p \xrightarrow{x(y)} p' \qquad q \xrightarrow{\overline{x}(y)} q'}{p \mid q \xrightarrow{\tau} (\nu\, y)(p' \mid q')}$$

Table 3.6: Late semantics of π-calculus

Indeed, $q \xrightarrow{\overline{x}(y)} \mathbf{0}$ *is the unique transition that can be inferred for* q *because, output on* z *is prevented by the restriction that violates side conditions of rules* $[res]$ *and* $[open]$ *and therefore,* $\{(p,q),(\mathbf{0},\mathbf{0})\}$ *is a bisimulation relation according to Definition 3.1.1. However, if we discard condition* $\mathrm{bn}(\mu) \cap \mathrm{fn}(p,q) = \emptyset$ *we could chose* w *for the newly generated name of* p *that does not fit for* q *because* w *is not new for it. This would prevent* q *to match the transition* $p \xrightarrow{\overline{x}(w)} \mathbf{0}$.

3.1.3 Late semantics

Late π-calculus, is an alternative semantics given in [130]. The main difference between late and early semantics is "the moment" at which input names are instantiated. Rule $[in]$ in Table 3.5 states that y is substituted for z when the input prefix is encountered. On the contrary, the late semantics instantiates it only when a synchronization effectively takes place.

We outline the late semantics of π-calculus "by difference" with respect to the early semantics reported in Section 3.1.2. The *late actions* that an agent can perform are defined as:

$$\mu ::= \tau \quad | \quad x(y) \quad | \quad \bar{x}y \quad | \quad \bar{x}(y).$$

The only difference with respect to the labels of the early semantics is the *bound input* action. The parenthesis should remind that y does not represent a name; it will be used as a "placeholder" that indicates where the effectively received name should be substituted in the continuation. Hence, we have that $\mathrm{fn}(x(y)) = \{x\}$ and $\mathrm{bn}(x(y)) = \{y\}$.

The transition rules remain the same of the early semantics apart from those reported in Table 3.6. The reader may notice that rule $[in^l]$ simply declares that an input action can be performed by a process without instantiating the formal parameter y to any actual value. When a free output action (rule $[comm^l]$) will synchronize on name x, the actual name z will be substituted for y in the continuation of the input process. Rule $[close^l]$ is similar but it does not require any instantiation because bound input and output transitions can always be renamed such that the bound names are turned into the same name.

An interesting exercise is to compare the natural bisimulation relation that arises from the late semantics.

Definition 3.1.2 (Late bisimulation) *A binary relation $\mathcal{R}$ over π-agents is a* late bisimulation *if, whenever $p\mathcal{R}q$ then*

- *for each $p \xrightarrow{\mu} p'$ with $\mu \neq x(y)$ and $\mathrm{bn}(\mu) \cap \mathrm{fn}(p,q) = \emptyset$, there is some $q \xrightarrow{\mu} q'$ such that $p'\mathcal{R}q'$;*

- *for each $p \xrightarrow{x(y)} p'$ with $y \notin \mathrm{fn}(p,q)$ there is some $q \xrightarrow{x(y)} q'$ such that, for all $z \in \mathcal{N}$, $p'[^z/_y]\mathcal{R}q'[^z/_y]$*

Two π-agents are late bisimilar *whether there is a late bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.*

The first thing to remark is that for non-input transitions, late and early bisimulation are defined in the same manner, namely, the first clause of Definition 3.1.2 is the same of the one in Definition 3.1.1. For input actions definitions differ each other; indeed, late bisimulation is stronger than early bisimulation because if $p \xrightarrow{x(y)} p'$ the choice of a transition $q \xrightarrow{x(y)} q'$ must not depend on the received name z. On the contrary, for the early semantics, we choose a free input transition of q depending on the received name in the transition of p.

Example 3.1.2 *Let us consider the following π-agents:*

$$q = x(y).\tau.\mathbf{0} + x(y).\mathbf{0} \qquad \text{and} \quad p = q + x(y).[y = z]\tau.\mathbf{0}.$$

Intuitively, p and q are early bisimilar because p can trivially mimic all transitions of q and q can mimic input transitions of the further addend of p because when y is substituted for z, we can choose the transition $q \xrightarrow{xz} \tau.nil$, otherwise, we choose $q \xrightarrow{xw} \mathbf{0}$ for the remaining transitions. In other terms, relation

$$\mathcal{R}_e = \{(p,q), ([z = z]\tau.\mathbf{0}, \tau.\mathbf{0})\} \cup \{([y = z]\tau.\mathbf{0}, \mathbf{0}) : y \neq z\}$$

is an early bisimulation that relates p and q.

On the other hand, p and q are not *late bisimilar because transition $p \xrightarrow{x(y)} [y = z]\tau.\mathbf{0}$ cannot be matched by q. Indeed, according to the late semantics, there are only two possible transitions that we might choose either $q \xrightarrow{x(y)} \mathbf{0}$ or $q \xrightarrow{x(y)} \tau.\mathbf{0}$. In the former case, Definition 3.1.2 requires that $([y = z]\tau.\mathbf{0})[^w/_y]$ and $\mathbf{0}[^w/_y]$ must be bisimilar for any w, which is not the case when w is z; whereas, in the latter case $([y = z]\tau.\mathbf{0})[^w/_y]$ is bisimilar to $\tau.nil[^w/_y]$ only when $w = z$.*

3.1.4 Variants of π-calculus

Several variants of the π-calculus have been proposed to study many aspects of concurrent and distributed systems. Since π-calculus has been widely used for modeling many facets of concurrent and distributed computation, pretending to give a complete list of citations would be too much ambitious. We focus here on some variants of π-calculus that are more closely related to our dissertation.

Some presentations of the calculus adopt *replication* [128], usually written as $!p$, in place of recursion. Process $!p$ can be intuitively explained as infinite copies of p in parallel. As far as expressiveness is concerned, the two methods for expressing infinite behaviours are equivalent (at the cost of additional silent actions). However, replication complicates the identification of a syntactic class of *finitary* agents. An agent is finitary if the number of parallel components of all its derivatives is bound. It is not decidable whether an agent is finitary or not, but it is possible to find syntactic conditions that ensures it. In the case of π-calculus with recursion, agents that do not have parallel composition in recursive definitions are finitary. Those agents are called *finite control agents* [54].

The *asynchronous* π-calculus [99, 24, 7] is a simple variant of the π-calculus where asynchrony is achieved by imposing the void continuation to the output actions. In other words, the (synchronous) π-calculus uses both input and output actions as prefixes while its asynchronous counterpart does not allow output prefixes: Outputs are processes of the form $\overline{x}y.\mathbf{0}$. Although from a theoretical point of view asynchronous π-calculus is less expressive than its synchronous version [144], it is still enough expressive in practice [99], and, in many respects, more adequate for modeling distributed computing.

The `join-calculus` [81] is an "extended subset" of asynchronous π-calculus which combines the three operators for input, restriction and replication into a single operator, called *definition*, that has the additional capability of describing atomic *joint* reception of values from different communication channels. The Distributed `join-calculus` [82] adds abstractions to express process distribution and process mobility.

Another linguistic extension is the introduction of polyadicity introduced in [128]. The polyadic version of the allows one to send and receive tuples of names instead of one single name along channels.

A significant variation of polyadic π-calculus is constituted by the *Fusion Calculus* introduced in [146]. The interesting aspect of Fusion Calculus is the complete symmetry between input and output actions. Indeed, input prefixes do not bind their object names. The effect of the synchronization of complementary actions is that object names are (globally) identified as shown in the communication rule below

$$\frac{p \xrightarrow{x\tilde{y}} p' \qquad q \xrightarrow{\overline{x}\tilde{z}} q'}{p \mid q \xrightarrow{[\tilde{y}=\tilde{z}]} p' \mid q'}$$

where $[\tilde{y} = \tilde{z}]$ stands for $[y_1 = z_1, ..., y_n = z_n]$.

Another extension of the polyadic calculus is *Distributed* π-calculus [94, 154]. This extension defines an explicit notion of locality that also affects channels that are *allocated*. Distributed π-calculus models distributed computations and access control policies. Locations reflect the idea of having administrative domains and located channels can be thought of as channels under the control of certain authorities. Moreover, distributed π-calculus provides a form of process mobility because processes can move from through localities.

3.2 Ambient Calculus

The syntax and the reduction semantics of Ambient calculus [39] is given below. The calculus relies on the notion of *ambient* that can be thought of as a bounded environment where processes interact. An ambient has a name, a collection of local agents and a collection of subambients. Ambients can be moved as a whole under the control of agents which are confined to ambients. Processes use *capabilities* for controlling interaction.

We do not consider synchronization and restriction, and replication is replaced by (guarded) recursion.

3.2.1 Syntax of Ambient

Ambient calculus proposes a different approach for modeling localities and migration of processes. An ambient system can be thought of as a number of localities, called *ambients*, that contain running processes or inner ambients, so realizing a hierarchical structure. By exercising movement capabilities, a process can pilot its surrounding ambient through the structure of the hierarchy. The syntax of Ambient is

Definition 3.2.1 (Ambient Syntax) *Let N be an infinite set of names ranged over by $a, b, c, ...$, $n, m, p, r...$; let $X, Y, Z, ...$ be process variables.*

$$\begin{array}{lcl} M & ::= & in\ n \ \mid \ out\ n \ \mid \ open\ n \\ P, Q & ::= & \mathbf{0} \ \mid \ n[P] \ \mid \ M.P \ \mid \ P|Q \ \mid \ rec\, X.\, P \ \mid \ X \end{array}$$

We assume that X is guarded *by M in $rec\, X.\, P$.*
We denote with Proc the set of the Ambient calculus processes.

We consider a fragment of Ambient without communication and restriction. Ambient processes use *capabilities* for controlling interaction. Indeed, by using capabilities, an ambient can allow other ambients to perform certain operations over it without having to reveal its actual name (which would give a lot of control over it). A name n is a capability to enter, exit or create a new copy of an ambient named

n. Capability *in* n serves for entering into ambient n, *out* n for exiting out of n and *open* n for opening up n.

A process is the void process $\mathbf{0}$, a process $n[P]$ obtained by wrapping P in an ambient n, a *sequential* process $M.P$, the parallel composition of two processes $P|Q$, the recursive process $rec\, X.\, P$ or a process variable X. Process $n[P]$ is an ambient with name n and process P running inside. Nothing prevents the existence of two or more ambients with the same name. Process $M.P$ executes the action corresponding to capability M and then behaves like P.

Also for Ambient calculus we can define a structural equivalence that disregard some syntactical aspects of terms.

Definition 3.2.2 (Ambient Structural equivalence) *The semantics of the Ambient calculus relies on the structural equivalence defined by the following rules:*

1. *The parallel operator* $\_|\_$ *is associative, commutative and* $\mathbf{0}$ *is its identity;*
2. $\dfrac{P \equiv Q}{M.P \equiv M.Q} \qquad \dfrac{P \equiv Q}{n[P] \equiv n[Q]}$;
3. $rec\, X.\, P \equiv rec\, Y.\, P[^{Y}/_{X}]$, *if* $Y \notin fv(P)$;
4. $rec\, X.\, P \equiv P[^{rec\, X.\, P}/_{X}]$.

The usual algebraic properties of the parallel composition and the $\mathbf{0}$ process are assumed (rule 1); structural equivalence is preserved by prefixing and ambient wrapping (rule 2); process variable X is bound in $rec\, X.\, P$ and may be renamed (rule 3); finally, recursive terms can be arbitrarily unfolded (rule 4).

3.2.2 Ambient semantics

The semantics of Ambient calculus is given by the reduction relation detailed in the following definition:

Definition 3.2.3 (Ambient Semantics) *The reduction relation* $\rightarrow \subseteq Proc \times Proc$ *is the relation inductively generated by the axioms and rules in table 3.7 and closed under the structural equivalence given in Definition 3.2.2.*

Even if simple, it suffers of the problems of reduction semantics discussed in Section 1.2 (see Observation 1.2.1) The first two axioms in Table 3.7 state that an ambient n can be driven by a sequential process inside it to exit the wrapping ambient ($out\ m.P$) or to enter a parallel ambient m ($in\ m.P$). The third axiom is relative to the *open* n capability: an ambient may be dissolved by an external process. Note that all the capabilities are "asynchronous", in the sense that the only condition under which they can be fired is the presence of a particular ambient.

We give an example of Ambient terms a relative reduction.

$$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$$

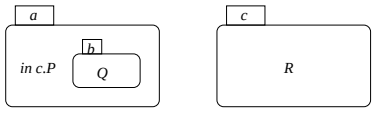
$$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$$

$$open\ n.P \mid n[Q] \rightarrow P \mid Q$$

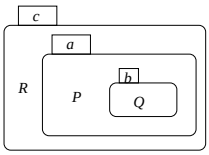
$$\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} \qquad \frac{P \rightarrow Q}{n[P] \rightarrow n[Q]}$$

Table 3.7: Ambient calculus reduction relation

Example 3.2.1 *Let consider the Ambient process together with an intuitive graphical representation given below:*

$a[\text{in } c.P \mid b[Q]] \mid c[R]$ 

Then the in-capability can be fired so that a is driven inside c in accordance with the following reduction

$a[\text{in } c.P \mid b[Q]] \mid c[R] \rightarrow c[R \mid a[P \mid b[Q]]]$ 

3.3 KLAIM

KLAIM [57] is an asynchronous higher–order process calculus which extends the Linda [86, 40] coordination paradigm (processes communicate via a shared multiset of tuples) to distributed and mobile processes. KLAIM can also be seen an experimental kernel programming language specifically designed to model and program distributed concurrent applications with code mobility. KLAIM naturally supports a *peer-to-peer* programming model where interacting peers (sites, in KLAIM terminology) cooperates to provide common sets of services.

KLAIM extends the basic Linda model by handling multiple distributed tuple spaces. Tuple spaces are placed on *sites* that are part of a *net*. Each site contains a single tuple space and processes in execution, and can be accessed through name which uniquely identifies it within the net. Hence, expression 'site' may be consistently used in place of 'site name', without generating misunderstandings.

Here we consider a version of KLAIM that differs from [57, 58, 59] and we will discuss the main differences between our KLAIM dialect and the earlier proposals in Section 3.4.

Fields and Tuples	f	$::=$	v	$\mid$	s
	t	$::=$	f	$\mid$	f, t
Template Fields and Templates	F	$::=$	f	$\mid !x \mid !u$	
	T	$::=$	F	$\mid$	F, T

Table 3.8: Tuple syntax

3.3.1 KLAIM syntax

Basically, KLAIM is a variant of the asynchronous π-calculus whose actions are the Linda primitives enriched with information about the addresses of the sites where processes and tuples are allocated. There are three types of values: basic values, sites (i.e. net addresses) and processes. Basic values are simply integers or strings or other elements of the elementary types that we do not specify. We shall use v, v_1, v_2 ... as generic basic values and x, y, z ... as generic variables for basic values. We let $\mathcal{S}$ be a set of *sites* and let $\mathcal{U}$ be a set of *site variables* ranged over by s and u, respectively; we use ℓ to denote a site or a site variable.

Observation 3.3.1 *We are not interested at formally specifying syntax for basic values. It may suffice to say that we can assume usual expression grammars normally defined for languages, the only constraint being that we assume variables to be basic values, namely element in $\mathcal{U} \cup \mathcal{S}$ are basic values.*

Table 3.8 reports the grammar for tuple constituting tuple spaces. A tuple space is a multiset of *tuples*; these are sequences of information items called *actual fields* (i.e. expressions or localities). Tuples are anonymous and content-addressable. *Pattern-matching* is used to select tuples in a tuple space by means of *templates*; these are containers of actual fields and *formal fields* (i.e. variables). Syntactically, a formal field is denoted with $!ide$, where ide is an identifier. The matching predicate, $match(T, t)$, is defined by the rules in Table 3.9. A template and a tuple match if

$$match(v, v) \qquad match(s, s)$$

$$match(!\,x, v) \qquad match(!\,u, s) \qquad \frac{match(F, f) \quad match(T, t)}{match((F, T), (f, t))}$$

Table 3.9: Matching rules

they have the same number of fields and corresponding fields match: a formal field matches any value of the same type, and two actual fields match only if they are identical. For instance, the template ("*foo*", $!x$) matches with the tuple ("*foo*", 1+2). After matching, the variable of a formal field gets the value of the matched field: in the previous example, x will contain the value 3.

Nets	N	::=	$\mathbf{0}$	*Empty net*
		|	$s :: P$	*Single site*
		|	$N_1 \parallel N_2$	*Net composition*
Actions	γ	::=	$\mathbf{in}(T)@\ell$	*Input*
		|	$\mathbf{read}(T)@\ell$	*Read*
		|	$\mathbf{eval}(P)@\ell$	*Process creation*
		|	$\mathbf{new}(u)$	*Site creation*
Processes	P	::=	$\mathbf{0}$	*Null process*
		|	$\mathbf{out}(t)$	*Output*
		|	$\gamma.P$	*Action prefixing*
		|	$P_1 \mid P_2$	*Process composition*
		|	$A\langle\widetilde{\ell}, \widetilde{v}\rangle$	*Process invocation*

Table 3.10: KLAIM syntax

Observation 3.3.2 *For simplicity, we assume that templates are* linear *which means that different formal fields of T use different variables and if $!x$ ($!u$) is a formal field of a template T then the only occurrence of x (of u) in T is the occurrence of the formal field . For instance, if we consider*

$$!x, x+1 \qquad !x, !x$$

are not linear templates, whereas $y, !x, 1$ is a linear template.

The syntax of the calculus is reported in Table 3.10. A net can be an empty net $\mathbf{0}$, a single site or the parallel composition of two nets N_1 and N_2 with disjoint sets of sites. A site $s :: P$, will be identified by its name s, where P is the processes running at s. It is of course possible that P, the process running at s, is the parallel composition of many other processes. The tuple space of s is modeled as a bunch of processes $\mathbf{out}(t)$ allocated at s.

Processes can perform four basic, possibly remote, operations over sites. $\mathbf{in}(T)@\ell$ looks in the tuple space located at ℓ for a tuple that matches T. Whenever the matching tuple t is found, it is removed from the tuple space. The values of the fields of t are assigned to the corresponding formal fields of T and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. $\mathbf{read}(T)@\ell$ differs from $\mathbf{in}(T)@\ell$ only because the matching tuple is not removed from the tuple space located at ℓ.
$\mathbf{eval}(P)@\ell$ spawns process P for execution at site ℓ.

New sites can be created through the operation $\mathbf{new}(u)$ and then accessed via the site variables u. This operation is not indexed with a site identifier because it is always executed at the current execution site.

Processes are built from the basic operations by using standard operators borrowed from process calculi: *null process*, *action prefixing*, *parallel composition* and *process invocation*. Processes can be defined parametrically by equations of the form $A(\widetilde{u},\widetilde{x}) \stackrel{\text{def}}{=} P$, where A is a *process identifier* and P is a process which may contain recursive calls (with the obvious parameter passing substitution). For each process identifier A there exists a *single* defining equation.

Variables occurring in process terms can be bound by action prefixes and process equations. More precisely, prefixes $\mathbf{in}(T)@\ell.\_$, $\mathbf{read}(T)@\ell.\_$ and $\mathbf{new}(u)$ act as binders for variables in the formal fields of t and for u, respectively. Process identifier definition $A(\widetilde{u},\widetilde{x}) \stackrel{\text{def}}{=} P$ is considered as a binder for variables $\{\widetilde{u},\widetilde{x}\}$. Albeit in KLAIM variables and names (values or sites) are distinguished, we continue to use $\mathrm{fn}(\_)$, $\mathrm{bn}(\_)$ and $\mathrm{n}(\_)$ for free, bound and variables of action, processes or nets.

Two processes that differ for α-renaming will be considered structurally equivalent. Moreover, we assume the usual monoidal laws for $|$ and $\|$ (the neutral elements being $\mathbf{0}$ and $\mathbf{0}$, respectively). We write $P \equiv Q$ to denote that P is structural equivalent to Q and, similarly, $N \equiv N'$ denotes the structural equivalence between nets N and N'.

3.3.2 KLAIM semantics

The operational semantics of the language is defined by the labelled transition system in Tables 3.11, 3.12 and 3.13. The semantics is given in terms of a transition relation, $\succ\!\!\longrightarrow$, that describes possible net evolutions (see Table 3.13). Relation $\succ\!\!\longrightarrow$ relies on:

- a labelled transition $\stackrel{a}{\longrightarrow\!\!\!\!\!\rightarrow}$, that describes process intentions to perform specific operations (see Table 3.11). Labels have the form $a(\_)$ where a is an element of the set $\{\mathbf{n},\mathbf{r},\mathbf{o},\mathbf{i},\mathbf{e}\}$ (reminiscent of the KLAIM actions) and its argument depends on the action;

- a labelled relation $\stackrel{a}{\Longrightarrow\!\!\!\!\!\rightarrow}$, that accounts for the intention of allocated processes, i.e. processes running on specific sites. In particular this relation considers the availability of resources (i.e. tuples and sites) in nets (see Table 3.12). Labels in Tables 3.12 (and 3.13) can be "allocated", namely they can be decorated with a site name or they can be of the form $s :: P$ which states that site $s :: P$ is available in the net.

We briefly comment on the rules in Tables 3.11. Rule (TUPLE) corresponds to signal the presence of a tuple. Prefixes (IN) and (READ) give account of the intention of a process that wants to access the tuple space located at s. (EVAL) is the rule for

$$\mathbf{out}(t) \xrightarrow{\mathbf{o}(t)} \mathbf{0} \qquad \text{(Tuple)}$$

$$\mathbf{in}(T)@s.P \xrightarrow{\mathbf{i}(T,s)} P \qquad \text{(In)}$$

$$\mathbf{read}(T)@s.P \xrightarrow{\mathbf{r}(T,s)} P \qquad \text{(Read)}$$

$$\mathbf{eval}(P)@s.Q \xrightarrow{\mathbf{e}(P,s)} Q \qquad \text{(Eval)}$$

$$\frac{s \notin \mathrm{n}(P)}{\mathbf{new}(u).P \xrightarrow{\mathbf{n}(s)} P[{}^{s}/_{u}]} \qquad \text{(Newloc)}$$

$$\frac{P[{}^{\widetilde{\ell},\widetilde{v}}/_{\widetilde{u},\widetilde{x}}] \xrightarrow{a} Q}{A\langle \widetilde{\ell}, \widetilde{v}\rangle \xrightarrow{a} Q} \; A(\widetilde{u},\widetilde{x}) \stackrel{def}{=} P \qquad \text{(PrDef)}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1|P_2 \xrightarrow{a} P_1'|P_2} \; \mathrm{bn}(a) \cap \mathrm{fn}(P_2) = \emptyset \qquad \text{(PrComp)}$$

Table 3.11: Process Semantics

spawning a process on a remote site s'. Rule (Newloc) generates a new site; note that the new created site is substituted for u in the continuation process. Finally, the last two rules are standard and similar to the corresponding π-calculus rules.

Table 3.12 details the behaviour of processes are running in a net and considers the resource availability. In particular, rule (Site) signals the presence of site $s :: P$ in the net, as well as (NetOut) signals the availability of a given tuple at site s. Continuation of rule (Site) is the empty net; at a first look, it can seem strange that s "disappear". However, rule (Site) is used only for determining the transition of rule (NEv) in Table 3.13 where the net semantics of **eval** prefix is given and the site "reappear" in the net. (NetComp), (comp) and (CompNew) lift the intention of processes at the net level. Note that last condition on the rule ensures that name captures are avoided by means of α-renaming.

Table 3.13 lists the rules for the net reduction semantics. (NEv) is the rule for executing an **eval** prefix. When a process wants to spawn P' on s, it is checked if site s is present or not. In the first case, P' is put in parallel with the processes already allocated at s, otherwise the **eval** prefix is blocked. Rule (NIn) synchronizes **in** and **out** actions provided that the respective template and tuple match. In this case, the formal variable are substituted in the continuation with the respective values in the tuple. Rule (NRead) is similar, however, in this case, it is necessary further machinery in order to avoid remotion of output tuple. Finally, (NNew) creates a new site that initially has no allocate process.

$$s :: P \xrightarrow{s :: P} \mathbf{0} \qquad \text{(Site)}$$

$$\frac{P \xrightarrow{\mathbf{o}(t)} Q}{s :: P \xrightarrow{\mathbf{o}(t)@s} s :: Q} \qquad \text{(NetOut)}$$

$$\frac{P \xrightarrow{a} Q \quad a \neq \mathbf{o}(t)}{s :: P \xrightarrow{a} s :: P'} \ \mathrm{bn}(a) \cap \mathrm{fn}(N_2) = \emptyset \qquad \text{(NetComp)}$$

$$\frac{N_1 \xrightarrow{a} N_1' \quad a \neq \mathbf{n}(s)}{N_1 \parallel N_2 \xrightarrow{a} N_1' \parallel N_2} \qquad \text{(Comp)}$$

$$\frac{N_1 \xrightarrow{\mathbf{n}(s)} N_1'}{N_1 \parallel N_2 \xrightarrow{\mathbf{n}(s)} N_1' \parallel N_2} \ s \notin \mathrm{n}(N_1 \parallel N_2) \qquad \text{(CompNew)}$$

Table 3.12: Located Processes Semantics

3.4 Related Works

This section tries to comments variants or other proposals of the presented process calculi. Since, works related to the π-calculus have been already discussed in Section 3.1.4, we limit ourselves in considering those proposal that can most closely be ascribed to families of calculi related to Ambient or Klaim.

Ambient Remarks Many aspects related to distributed programming and Ambient have been investigated by several researchers.

A first challenging problem is the definition of a label transition system that faithful represents the reduction semantics of the calculus. Some basic researches have tried to equip an interactive semantics for the Ambient calculus. A LTS operational semantics for ambients has been defined by Gordon and Cardelli in an unpublished note [38]. It requires the introduction in the calculus of co-actions, abstractions, concretions and outcomes. As far as we know, the bisimilarity abstract semantics based on this operational semantics has not been compared with the reduction semantics. A modal logic for reasoning about spatial and temporal aspects of ambients has be developed in [32, 33, 34], which is equipped with specialized modalities to deal with the spatial and the temporal dimensions of global computing. Sewell [165] introduces a technique to develop an LTS-based semantics from a reduction semantics; however the resulting transition semantics exploits arbitrary contexts and, moreover, it is not inductive on process operators. Recently, an LTS semantics for Ambient calculus has been devised in [13]. The proposed methodol-

$$\frac{N_1 \xrightarrow{\mathbf{e}(P',s)} N_1' \qquad N_1' \xrightarrow{s :: P} N_2}{N_1 \rightarrowtail N_2 \parallel s :: P|P'} \quad (\text{NEv})$$

$$\frac{N_1 \xrightarrow{\mathbf{o}(t)@s} N_1' \qquad N_1' \xrightarrow{\mathbf{i}(T,s)} N_2 \qquad match(T,t)}{N_1 \rightarrowtail N_2[^{t'}/_t]} \quad (\text{NIn})$$

$$\frac{N_1 \xrightarrow{s :: P} N_1' \qquad P \xrightarrow{\mathbf{o}(t)} Q \qquad N_1' \xrightarrow{\mathbf{r}(T,s)} N_2 \qquad match(T,t)}{N_1 \rightarrowtail N_2[^{t'}/_t] \parallel s :: Q \mid \mathbf{out}(t)} \quad (\text{NRead})$$

$$\frac{N_1 \xrightarrow{\mathbf{n}(s_2)} N_2}{N_1 \rightarrowtail N_2 \parallel s_2 :: \mathbf{0}} \quad (\text{NNew})$$

Table 3.13: Net Semantics

ogy is a general framework for the analysis of open systems and can be exploited for reasoning on spatial and temporal properties of Ambient processes.

Graph-based semantics of Ambient have been proposed in [85]. In [85] a translation of Ambient into particular hypergraphs is given. Dynamic behaviour of Ambient processes is obtained using the *Double Pushout* approach [50].

Recently, to cope with the interferences that arise when implementing interaction protocols, a variant of the Ambient calculus have been proposed [113] that makes use of *co-actions*. In such a variant, each Ambient action has a complementary action and a reduction can occur only whenever two complementary actions do synchronize. A type system guarantees that well-typed terms will never generate undesired interferences at execution time.

In [37, 36] Ambient has been proposed as a foundational model for semi-structured data. In particular, a query language for semi-structured has been defined; the language is based on the Ambient logic and can "manipulate" semi-structured data modeled as Ambient processes.

Ambient calculus has also been extended for expressing issues related to resource control in reconfigurable parallel systems [168]. This Ambient extension is equipped with a type system that makes possible to control resource accesses.

An experimental implementation of the Ambient calculus, called Ambit, can be found at `http://www.luca.demon.co.uk/Ambit/Ambit.html` and consists of just a `Java` applet. A distributed implementation of the calculus is presented in [83]. The implementation relies on a formal translation of Ambient in Distributed Join Calculus and uses `JoCaml` as implementation language.

KLAIM **remarks** Originally, KLAIM distinguished between *logical* and *physical* localities [57, 58, 59]. This distinction is a peculiarity of KLAIM and allows to separate two concerns, namely programming the web and allocate an application on the physical sites. A programmer can think of a link as a symbolic link disregarding of the real allocation of resources. KLAIM processes are executed afterward an allocation phase that maps them onto the sites of a net has take place. Once the application is allocated, an *allocation environment* is associated to any physical site. This is an interesting mechanism if related to code mobility provided by KLAIM because permits to use dynamic scoping and, therefore, computations depend and their execution sites.

We avoid such distinction essentially for presentation purposes. The fragment detailed in 3.3 does not consider allocation environment because we want to focus on an extension with constructs that explicitly build connections between sites that must be accessed in order to make different sites to cooperate. We will provide a translation of this KLAIM variant in the graph model presented in 4.2. We remark that allocation environments and dynamic scoping can be easily added to the calculus and the mapping to hypergraphs at the cost of complicating productions that would obscure the main advantages of our translation.

A minor consequent difference is also the fact that mobility is obtained in our variant only by means of the **eval** construct, while, in the original calculus, processes can also be exchanged in tuples. The reason is that in the original calculus processes exchanged in tuples offer the opportunity of having also static scoping. Indeed processes in tuples are *closures*, i.e. a process equipped with an allocation environment that is used for resolving the symbolic links referred in the process. This allows one to move processes together the original environment that is used for resolving the symbolic link at execution time. Since we do not have allocation environment, this would not be very useful in our calculus.

Many aspects of WAN programming have been considered in various KLAIM dialect; in [59] KLAIM has been equipped with a type system for controlling accesses to resources; the type system ensures that well-typed nets never give rise to unauthorized accesses. Hierarchical networks have been considered in [18], whereas [19] equips KLAIM with primitives for programming dynamic changes of network connectivity. An experimental implementation of KLAIM, called X-KLAIM, can be downloaded at `http://rap.dsi.unifi.it/klaim.html`. The implementation consists of two layers: the X-KLAIM compiler and the intermediate language KLAVA that is obtained by extending the JAVA language [9] with a new package, called `Klava`. The `Klava` package [17] contains all the classes which implement the X-KLAIM runtime system and operations.

3.5 Security Issues

Computations of WAN applications consist of communications in an environment where communication links are not under the control of a central (trusted) authority, namely network links are "public". Hence, it is possible for an *intruder* to "compromise" the intended interactions by accessing to the public communication structure. An intruder can "read" all the information exchanged by components of WAN applications, steal messages directed to other receivers or substitute them with fake messages.

Cryptography and *security protocols* have been introduced to enhance, as far as possible, robustness of WAN systems and applications with respect to malicious intruders. Cryptography is used to provide authentication, to distribute cryptographic keys for new conversations or, more generally, to communicate secrets through non-trusted *media*.

Protocol design is guided by the security properties that the protocol aims at guaranteeing. Indeed, usually a protocol guarantees some properties but it cannot ensure *any* property. However, a good protocol design must take into account many aspects; for instance, the overall hypothesis is that *principals*[2], operate in an open hostile environment and some of the participants may be untrusted; even if the protocol guarantees that some secret information, as encryption keys, cannot be leaked, an attacker might obtain those informations in other ways and, therefore, a good protocol should minimize the effects of such events.

Even when security protocols have been carefully developed by experts and reviewed carefully by other experts, they might be later found to have flaws that make them "attackable". A well known example is the Needham-Schroeder public key protocol [138] found to be flawed by Lowe [115] after being supposed correct for many years.

For such reasons, methods to formally reason about and analyze protocols are required. Analyzing security protocols consists mainly in two complementary activities. The first is to find flaws in those protocols that are not correct. The second is to establish convincingly the correctness of those that are. These activities are interrelated, because the discovery of a flaw may suggest an altered protocol that we may wish to prove correct, and because a failure to prove the correctness of a protocol may suggest a particular flaw [69].

In Chapter 8.1 we will consider topics related to security.

[2] A principal is a "regular" participant in protocol sessions.

Part I

A Model for Declarative WAN Programming

Abstract

This part of the thesis proposes a declarative approach to WAN programming. The approach is based on a graphical calculus that allows one to specify components and their synchronization requirements as *hyperedges* and constraints on their interface, respectively. Computations are modeled by means of a rewriting mechanism that allows hyperedge replacement. A hyperedge is replaced when constraints on its synchronization interface are satisfied by the other connected hyperedges. This amounts to *mobility* of components, that dynamically can change their connections.

We feel that on the one side the generality of the approach can be tamed and adapted to the needs of the various layers of applications, more powerful primitives being made available to upper layers. In particular, we exploits the graphical calculus as an intermediate language for representing Ambient and show how the mechanisms of the calculus suggests new Ambient-like primitives. On the other side, some important network technologies actually require the solution of global constraints, like modifying local router tables according to the routing update information sent by the adjacent routers. With respect to this issue, we propose a variant of KLAIM that permits to express attribute on connections between sites of the net and show how its translation into our graphical calculus accounts for managing network connections and modeling application declared routing requirements.

Finally, a methodology where the graphical calculus is used for specifying and refining systems is pointed out.

Chapter 4

Hypergraphs

Abstract

Graph-based techniques can be usefully adopted for modeling internetworking systems. Indeed, hyperedges can be used to represent components and nodes to model the network environment of components. Edges sharing a node means that the corresponding components may interact by exploiting network communication infrastructure.

Graph synchronization adds to network awareness the ability of dealing with the temporal dimension of computations. Graphs synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. Nodes can be exchanged during synchronizations, hence constraint solving must encompass unification in order to fuse node. We propose a new edge rewriting mechanism that allows interconnection modification.

This chapter extends the notion of *hypergraph* as presented in [97]. First hypergraph definition is given, then *hypergraph rewriting systems* are introduced; finally, hypergraphs operational semantics is defined.

Contents

4.1 Graph Grammars

Graph grammars [158] have been proposed as an extension of grammars of strings. Grammars of strings specify how strings can be rewritten according to a set of *productions*, which are rules of the form $\alpha \longrightarrow \beta$, where α and β are strings over a fixed alphabet (containing terminal and non-terminal symbols). Depending of the form α and β different classes of grammars can be defined. For instance, the well-known contex-free grammars are defined as the grammars having productions $\alpha \longrightarrow \beta$ where α is a string made of a single non-terminal symbol.

Many similitudes can be traced between grammars of strings and graph grammars. Productions of a graph grammar specify how graphs can be rewritten according to some productions; they can be more complex than productions of string grammars. For the sake of simplicity, let us assume that productions of graph grammars have the form $L \longrightarrow R$ where L and R are graphs. The concept of "context-freeness" can be also found in graph grammars: More precisely, productions with a left-hand side which is either a node or an edge confer a "context-free" flavour to graph grammars. Indeed, such productions do not consider the "surroundings" of their left-hand sides.

Other approaches to graph rewriting have be proposed, as well. In particular, we mention the algebraic approaches: The *Double Push Out* (DPO) and the *Single Push Out* (SPO). Since DPO and SPO replace sub-graphs instead of nodes or edges, the analogy with string grammars is that they are not contex-free grammars, indeed, they generalize the type-0 Chomsky string grammars. We will not discuss in detail all those approach (the interested reader is referred to [158] for complete presentation).

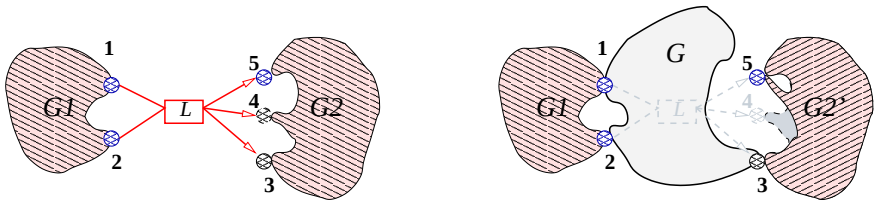
4.1.1 Hyperedge replacement: An informal introduction

All the approaches described in Section 4.1 can be extended to *hypergraphs* that will formally be defined in Section 4.2. For the moment it suffices to say that hypergraphs are graphs made of nodes and a particular type of edges called *hyperedges*. Essentially, a hyperedge may be thought of as being an edge connecting more than two nodes. Figure 4.1(a) depicts an hyperedge L relating five nodes.

In [43, 61] edge replacement and synchronization has been proposed as a mechanism for rewriting graphs. In [136] synchronized edge replacement has been related to distributed constraint satisfaction problems. In this dissertation we will introduce a new rewriting mechanism which extends the synchronized edge replacement proposed in [97].

As will be formally stated in the sequel, synchronizations are used to coordinate adjacent edges in a hypergraph. Indeed, synchronized hyperedge replacement will be exploited to constraining graph rewriting.

Figure 4.1 aims at giving an intuition of hyperedge replacement. The hyperedge L in Figure 4.1(a) is connected to $G1$ and $G2$ through its *external nodes* (or *at-*



(a) An hyperedge L (b) Replacement of L with G

Figure 4.1: Hyperedge replacement

tachment points) $\mathbf{1}, \ldots, \mathbf{5}$: Nodes **1** and **2** are attachment points of both L and $G1$, while nodes **3**, **4** and **5** are attachment points of both L and $G2$.

Figure 4.1(b) represents the hypergraph obtained by replacing L with hypergraph G. The dashed gray part of the Figure 4.1(b) represents the initial situation and is the part that disappears after the replacement of L has taken place.

The main things to remark are that

i. parts of the graph may not be involved in the rewriting, e.g. $G1$ is not modified after the transition;

ii. after the rewriting new nodes can appear; indeed, all nodes in G different from **1** and **2** are new nodes generated by the transition;

iii. some nodes can be "fused" after the transition; in Figure 4.1(b) node **4** is fused with **5**. As will be shown later, this amounts to *mobility* of components, that dynamically can change their connections. In fact, notice that, in Figure 4.1(b), the part of $G2$ that was connected to node **4** corresponds, in Figure 4.1(c), to the part of $G2'$ connected to node **5**.

The replacement is triggered by the fact that, all the conditions imposed by L at its attachment nodes are matched by all the other connected components.

We want to point out the motivations for using edge replacement instead of DPO or SPO approaches. The main reason is that the contex-free flavour of edge replacement allows us to exploit graph rewriting for defining a declarative approach to WAN programming. Intuitively, a component of a distributed system is represented by a edge connected to (several) nodes. Edge that share nodes can interact and coordinate their activities. Hence, nodes can be thought of as synchronization ports where communications take place.

This approach is "declarative" because the behaviour of each component is specified by declaring some condition on their synchronization ports indipendently from the other components. Once the system is built (by opportunely connecting its components) it will evolve according to the possible synchronizations of the edges. It should be clear that the same features are difficult to be achieved using DPO or

SPO. Indeed, productions involves complex sub-graphs and not single components. Therefore, it is difficult to specify the behaviour of a part of a system independently from other parts.

4.1.2 Hyperedge replacement: An informal example

A production rewrites a single edge into an arbitrary graph. Before giving the formal definition of edge replacement we give an intuitive description of our synchronized hyperedge replacement mechanisms.

Productions $p_1 : L_1 \to R_1$,..., $p_h : L_h \to R_h$ can be applied to a graph G yielding H if, for any $i = 1, ..., h$ there exists an occurrence of an edge labelled by L_i in G such that, for any external node of the edge that is subject to a condition in p_i, all the adjacent edges in G

- either have labels different from all L_j,
- or they are labelled by L_j (for some $j \neq i$) and the corresponding condition in p_j is in accordance with the synchronization algebra adopted.

In the former case the adjacent edge does not take part in the synchronization; while in the latter case they must agree according to the adopted synchronization policy. If the previous conditions hold, H is obtained from G by

1. removing the occurrence of L_1,...,L_h
2. embedding a fresh copy of R_i (for any $i = 1, ..., h$) in G and
3. coalescing external vertices of all R_i's with the corresponding attachment vertices of the occurrence of edge L_i.

This description can be naively regarded as a procedural way of obtaining hypergraph rewritings.

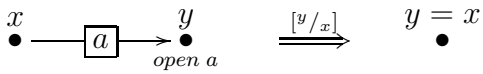
In order to give an intuition of synchronized hyperedge replacement with fusions, we give an informal account of how the open primitive of the Ambient calculus (see Section 3.2, page 48) can be simply modeled with a graph grammar. A detailed definition of productions for the Ambient calculus will be introduced in Chapter 5. For the moment we do not consider exchanging of nodes in the productions because we want to focus the attention of the reader on node fusions determined by the productions.

An ambient process $a[\,]$ and a capability *open* a can be represented by edges

where x is the attachment node for processes inside a, while y and z respectively are the attachment nodes of a and $L_{open\ a}$ to their surrounding environment.

Observation 4.1.1 *In the Ambient calculus, ambients (e.g. $a[\,]$) are used to model "localities", but are here represented by hyperedges connecting two nodes. In general, nothing prevents to represent localities with nodes instead of hyperedges. However, localities in calculi like Ambient (and* KLAIM*) act as* coordinators*, hence it is better to represent them as hyperedge and model their coordination activities with suitable productions.*

Let us consider the following production for a:

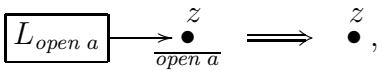


(4.1)

the left-hand side of production (4.1) represents edge a and the synchronization constraint on its external nodes; the substitution on the arrow maps nodes on the left-hand side into themselves and gives the fusions that must take place once the production is applied; the right-hand side is the graph that replaces edge on the left-hand side. Production (4.1) can be read as:

> An edge a connecting nodes x and y and representing an ambient process cancels itself and fuses x (the attachment node of its inner processes) on y (the attachment point toward the surrounding environment) provided that the environment satisfies the synchronization constraint *open* a.

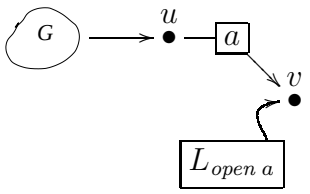
The production for $L_{open\ a}$ follows:



(4.2)

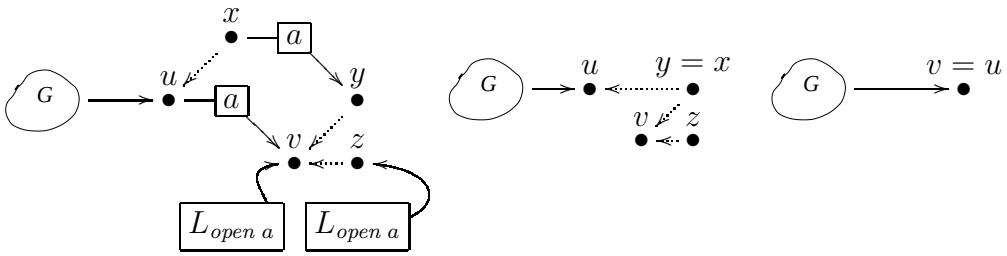
it simply states that edge $L_{open\ a}$ disappears when the surrounding environment satisfies constraint $\overline{open\ a}$ (we do not represent the identity substitution on the arrow).

Let us now consider the ambient term $a[Q] \mid open\ a.\mathbf{0}$ that can be represented with the following graph:



where G is the graph that corresponds to Q.

In order to synchronize productions (4.1) and (4.2), first we must find their occurrences on the correspondence with the external nodes in the graph, as reported in Figure 4.1.2(a) where the dotted lines represent the correspondence among the external nodes. Then, as shown in Figure 4.1.2(b), the edges in the graph are removed and fresh copies of the right-hand sides of the productions are added to the graph (in this case the rhs of the first production simply is the node y, while the



(a) Finding occurrences (b) Embedding rhs (c) Fusing nodes

Figure 4.2: Synchronized edge replacement

rhs of the second production simply is the node z). Finally, nodes are fused; notice that u and v are fused because they correspond to x and y respectively and those have been fused as prescribed by production (4.1).

We point out that the external nodes of edges a and $L_{open\ a}$ in the graph satisfy the requirements for replacing edges. Indeed, node u corresponds to x which has no condition in production (4.1) while v corresponds to y and z and the conditions imposed on them by the productions are complementary, hence they can be synchronized. As a final remark, notice that G is not involved in the synchronizations, however, after the synchronization G is attached to node v.

4.2 A Calculus of Hypergraphs

In the following we consider fixed a set of nodes $\mathcal{N}$ and a set of labels $\mathcal{L}$ ranked by natural numbers; $L : n$ denotes a label $L \in \mathcal{L}$ with rank n.

Hypergraphs are built out from *hyperedges* and nodes.

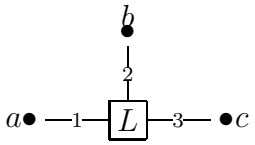
Definition 4.2.1 (Hypergraph) *A* Hypergraph *H over $\mathcal{L}$ is a five-tuple (V, E, att, lab, ext) where*

- *$V \subseteq \mathcal{N}$ is a finite set of nodes;*
- *E is a finite set of* hyperedges*;*
- *$lab : E \to \mathcal{L}$ is a* labelling function*;*
- *$att : E \to V^*$, where V^* is the set of sequence of nodes in V. For each $e \in E$, $att(e)$ are the* attachment nodes *(or* attachment points*) of e and $|att(e)| = n$ iff $lab(e) : n$;*
- *$ext \in V^*$ are the external nodes of H.*

A *hyperedge* e (or simply an *edge*) is an atomic item with a label $lab(e)$ and an associated sequence of attachment nodes $att(e)$. The number of attachment nodes

of an edge must be the rank of its label. We write $L(x_1, ..., x_n)$ to indicate an edge labelled L connected to the attachment nodes x_1,..., x_n.

Example 4.2.1 *Let $L(a, b, c)$ be an edge where $L \in \mathcal{L}$ and $L : 3$; we draw $L(a, b, c)$ as follows*



where wires connecting vertices a, b and c to L are called tentacles *and are indexed according to the sequence of attachment nodes.*

In drawings hypergraphs we will omit indexes of tentacles when they are clear from the context; moreover we depict external nodes with • while non-external nodes are drawn with ∘.

In the sequel, we represent hypergraphs by means of *syntactic judgments*:

Definition 4.2.2 (Syntactic judgments) *A* syntactic judgment *is a judgment of the form $\Gamma \vdash G$, where*

- *$\Gamma \subseteq \mathcal{N}$ is a finite set of nodes and*
- *G is a term generated by the following grammar*

$$G ::= nil \mid L(\vec{x}) \mid G \mid G \mid \nu\, y.G,$$

where $\vec{x}$ is a sequence of nodes, $L \in \mathcal{L}$ is such that $L : |\vec{x}|$ and $y \in \mathcal{N}$. We call terms G hypergraph terms.

The nodes in G which are in the scope of ν operator are called bound *nodes; let* $\mathrm{bn}(G)$ *and* $\mathrm{fn}(G)$ *respectively denote the set of the bound and* free *nodes of G (the nodes of G which are not bound).*

A judgment $\Gamma \vdash G$ is legal *if* $\mathrm{fn}(G) \subseteq \Gamma$.

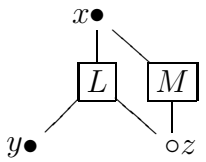
Productions in Definition 4.2.2 permits generating the empty graph (represented by *nil*), single edges (using $L(\vec{x})$) composing terms in parallel (via $G \mid G$) and hiding nodes (through $\nu\, y.G$).

Hereafter, we use 'graph' as a synonym of 'hypergraph' and omit curly brackets in judgments writing $x_1, ..., x_n \vdash G$ instead of $\{x_1, ..., x_n\} \vdash G$; moreover, given a sequence of nodes $\vec{x} = x_1, ..., x_n$, $\vec{x} \vdash G$ denotes $x_1, ..., x_n \vdash G$. We use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$. Similarly, we will write Γ_1, Γ_2 to state that the resulting set of names is the disjoint union of Γ_1 and Γ_2. Two judgments $\Gamma_1 \vdash G_1$ and $\Gamma_2 \vdash G_2$ are *disjoint* if $\Gamma_1 \cap \Gamma_2 = \emptyset$ and, $\Gamma_1 \vdash G_1 \otimes \Gamma_2 \vdash G_2$ denotes the judgment $\Gamma_1, \Gamma_2 \vdash G_1 \mid G_2$ provided that initial judgments are disjoint.

$(AG1)$	$(G_1 \mid G_2) \mid G_3 \equiv G_1 \mid (G_2 \mid G_3)$	
$(AG2)$	$G_1 \mid G_2 \equiv G_2 \mid G_1$	
$(AG3)$	$G \mid nil \equiv G$	
$(AG4)$	$\nu\ x.\nu\ y.G \equiv \nu\ y.\nu\ x.G$	
$(AG5)$	$\nu\ x.G \equiv G,$	if $x \notin \mathrm{fn}(G)$
$(AG6)$	$\nu\ x.G \equiv \nu\ y.G\{y/x\},$	if $y \notin \mathrm{fn}(G)$
$(AG7)$	$\nu\ x.(G_1 \mid G_2) \equiv (\nu\ x.G_1) \mid G_2,$	if $x \notin \mathrm{fn}(G_2)$

Table 4.1: Graphs structural congruence rules

Example 4.2.2 *Let us consider the judgment* $x, y \vdash \nu\ z.(L(y,z,x) \mid M(x,z))$, *where* $L : 3$ *and* $M : 2$*; a graphical representation of the judgment is*



Productions for G in Definition 4.2.2 are very similar to the respective productions for π-calculus grammar. Definition 4.2.3 gives the structural congruence rules for graphs. We take advantage of such congruence to avoid writing cumbersome parenthesis.

Definition 4.2.3 (Structural Congruence) *The* structural congruence *is the smallest binary relation* $\equiv$ *that obeys axioms in Table 4.1.*

Axioms $(AG1)$, $(AG2)$ and $(AG3)$ define associativity, commutativity and identity over *nil* for operation $|$, respectively. Axioms $(AG4)$ and $(AG5)$ state that the nodes of a graph can be restricted in any order and that restriction does not play any rôle on non-free nodes of a graph, respectively. Axiom $(AG6)$ deals with alpha conversion of hidden bound vertices, while $(AG7)$ tunes the interplay between hiding and the operator for parallel composition. Occasionally, because of axiom $(AG4)$, we will write $\nu\ X$, with $X = \bigcup x_i$, to abbreviate $\nu\ x_1.\nu\ x_2 \ldots \nu\ x_n$. Note that using the axioms, for any judgment we can always have an equivalent normal form $\Gamma \vdash \nu\ X.G$, with G a sub-term containing only parallel composition of edges. It is clear from Definition 4.2.3 that Γ and X can be made disjoint sets of nodes using the axioms and that nodes of G are included in $(\Gamma \cup X)$.

We will work with *well-formed judgments*.

$$\frac{}{x_1,\ldots,x_n \vdash nil}\,(RG1) \qquad \frac{\Gamma \vdash G_1 \qquad \Gamma \vdash G_2}{\Gamma \vdash G_1 \mid G_2}\,(RG3)$$

$$\frac{L : m \qquad y_1,\ldots,y_m \in \{x_1,\ldots,x_n\}}{x_1,\ldots,x_n \vdash L(y_1,\ldots,y_m)}\,(RG2) \qquad \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu\, x.G}\,(RG4)$$

Table 4.2: Well-formed judgments

Definition 4.2.4 (Well-Formed Judgments) *A judgment is* well-formed *if it is generated by applying the rules in Table 4.2 up to structural congruence.*

Rule (*RG1*) creates a graph with no edges and n isolated nodes and rule (*RG2*) creates a graph with n nodes and one edge labelled by L and with m tentacles (note that there can be repetitions among nodes in $\vec{y}$, i.e. some tentacles can be attached to the same node). Rule (*RG3*) allows to put together (using $|$) two graphs that share the same set of external nodes. Finally, rule (*RG4*) allows to hide a node from the environment.

The correspondence theorem expressing that well-formed judgments up to structural axioms are isomorphic to graphs up to isomorphism has been proved in [97]. This result allows us to consider syntactic judgments as hypergraphs; in the sequel, we will sometime refer to edges in a graph by saying 'edge $L(\vec{x})$' which is not formally correct, but the result proved in [97] permits this abuse of terminology.

We can extend concepts and definitions for ordinary graphs to hypergraphs. We need to give a formal definition for paths and cyclic graphs.

Definition 4.2.5 (Hyperpath) *A* hyperpath *is a term of the form $L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k)$ such that, for any $i = 1,..,k-1$, there exists a node $n \in \mathcal{N}$ such that L_i and L_{i+1} are connected to n. In this case we say that L_i and L_{i+1} are* adjacent *edges.*

Definition 4.2.5 states that two nodes are connected by a path whether the adjacent hyperedge on the path share at least one of their attachment nodes. This permits us to define *ambient graphs* (see Definition 5.1.2 on page 85) and paths on graphs defined in Chapter 6 where a path between two nodes obtained alternating two different labelled eges (namely *site* and *gateway* edges) represent a possible connection between two site of a KLAIM net.

Definition 4.2.6 (Hypercycles) *A* cycle *is a hyperpath $L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k)$ such that L_k and L_1 are adjacent.*

A term G is acyclic if, for any path $L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k)$, if there is a finite set of nodes X and a term G' such that $G \equiv (\nu\ X)(L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k) \mid G')$, then $L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k)$ is not a cycle.

Definitions 4.2.5 and 4.2.6 can trivially be extended to judgments and will be exploited in the sequel.

4.3 Hypergraph Rewriting

This section formalizes the hypergraph rewriting mechanism that we have informally described in Section 4.1. Synchronized edge replacement is obtained by graph rewriting combined with constraint solving. More specifically, we use *context-free* productions labelled with actions useful for coordinating the simultaneous application of two or more productions. Coordinated rewriting allows the propagation of synchronization all over the graph where productions are applied. Determining the productions to synchronize at a given stage corresponds to solving a distributed constraint satisfaction problem [136].

In [97] a synchronized hyperedge replacement mechanism has been defined. The main feature of this approach is that an hyperedge can be replaced when the conditions it imposes on its external nodes are in accordance with the conditions imposed by adjacent hyperedges and with the adopted synchronization policy.

Observation 4.3.1 *Each edge rewriting must synchronize its actions with one or more of its adjacent edges. Depending on the chosen synchronization algebra, the number of edges can vary.*

The main and well studied synchronization algebras are á la *Hoare (CSP [98], where $S(a, a) = a$) and* á la *Milner (CCS [127], where $S(a, \bar{a}) = \tau$) synchronizations [173]. The former takes two partner to synchronize trough complementary actions, while the latter requires that all participants of a synchronization perform the same action. Hereafter, we consider synchronizations* á la *Milner.*

Replacements can generate new nodes, exchange nodes and fuse them. However, the approach of [97] imposes a restriction on node fusion: Two nodes can be coalesced only if at least one of them is new, namely it is not possible to fuse two "old" nodes.

Our approach extends the proposal in [97] by permitting fusions without any restriction as described in the informal example of Section 4.1.2.

4.3.1 Productions

In the following, we assume fixed Act, a set of actions used for naming conditions imposed on the external nodes of hyperedges for constraining graph rewritings. Since we use Milner synchronizations, Act also has two furhter ingredients:

- a complementation operation $\bar{\cdot} : Act \to Act$ such that for any $a \in Act$, $\bar{\bar{a}} = a$;
- a distinguished *silent* action τ that denotes synchronizations.

Moreover, we assume that any label $a \in Act$ has an *arity*; we let $|\cdot| : Act \to \omega$ be the arity function on Act and, for any $a \in Act$, $|a| = |\bar{a}|$.

A *graph rewriting system*, $\mathcal{G} = \langle \Gamma_0 \vdash G_0, \mathcal{P} \rangle$, consists of an initial graph $\Gamma_0 \vdash G_0$ and a set of *productions*:

Definition 4.3.1 (Production) *Let $X \subseteq \mathcal{N}$ be the set $\{x_1, \ldots, x_n\}$ and L be an edge label with rank n. A* production *is a transition of the form*

$$X \vdash L(x_1, \ldots, x_n) \xrightarrow[\pi]{\Lambda} \Gamma \vdash G, \tag{4.3}$$

where

- $\Lambda \subseteq X \times Act \times \mathcal{N}^*$ *is a* set of constraints *which are triples $(x,\ a,\ \vec{y})$ such that $|a| = |\vec{y}|$. Λ is the graph relation of a partial function with (finite) domain X and codomain in $Act \times \mathcal{N}^*$.*

  Given Λ, we say that $\vec{y}$ are the nodes of the constraint *$(x, a, \vec{y}) \in \Lambda$ and $\Lambda(x)$ indicates $(a, \vec{y})$, while $\Lambda(x) \uparrow$ states that Λ does not imposes constraints on x (i.e. $(x', a, \vec{y}) \in \Lambda$ implies $x \neq x'$). We let $\mathrm{n}(\Lambda)$ denote the union of the nodes of the constraints in Λ.*

- *function $\pi : X \to X$ is a* fusion substitution*, namely*

$$\forall x_i, x_j \in X. \pi(x_i) = x_j \Rightarrow \pi(x_j) = x_j.$$

  A fusion substitution π induces an equivalence relation partition $\simeq_\pi$ over its domain X defined as $x \simeq_\pi x' \stackrel{def}{\iff} \pi(x) = \pi(x')$. Equivalence $\simeq_\pi$ partitions X into equivalence classes and each node $x \in X$ is mapped to the representative element *$\pi(x)$ of its class. The representative element y of a class is the unique element such that $\pi(y) = y$.*

  We impose a further condition on productions, indeed we require that $\mathrm{n}(\Lambda) \cap X \subseteq \pi(X)$; namely, the external nodes used in the synchronization must be representative elements according to $\simeq_\pi$.

- $\Gamma = \pi(X) \cup (\mathrm{n}(\Lambda) \setminus X)$;

- $\mathrm{fn}(G) \subseteq \Gamma$.

Nodes $x_1, \ldots, x_n$ are the attachment nodes of L to the surrounding environment, namely L can share nodes in X with other edges. Productions specify the constraints that the environment must satisfy in order to replace edges. Such constraints are imposed by Λ on the set X of external nodes of L; arities of actions must be equal to the number of nodes of the constraint. Λ associates actions and sequences of nodes to (some of the) external nodes of L. Intuitively, actions associated to attachment nodes constrain the possible rewritings of a graph; indeed, production (4.3) can be applied only if the actions on the external nodes synchronize with actions imposed

by the productions of adjacent edges according to the synchronization adopted policy. If $(x, a, \vec{y}) \in \Lambda$ then L can synchronize with edges in the environment that have a tentacle connected to x and satisfy condition a. Sections 4.3.1 will formally state how constraints are satisfied by means of synchronizations (for *á la* Milner synchronization).

Once constraints in Λ are satisfied, nodes must be coalesced according to fusion substitution π.

For any constraint $(x, a, \vec{y}) \in \Lambda$, $\vec{y}$ either contains nodes that appear in X or new nodes that will be present in $\Gamma \vdash G$.

Let us now consider the structure of the right hand side of judgment (4.4). Γ consists of the nodes which are image of $x_1, \ldots, x_n$ through π and the new nodes used in the synchronization, namely those nodes that appear in Λ and are not in X. In general, G may be any graph provided that $\mathrm{fn}(G) \subseteq \Gamma$.

Example 4.3.1 *Referring to Example 4.2.2, let us assume that the following production is given:*

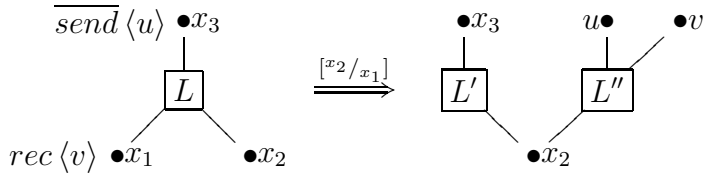
$$x_1, x_2, x_3 \vdash L(x_1, x_2, x_3) \xrightarrow[[x_2/x_1]]{\left\{\begin{array}{c}(x_3,\ \overline{send},\ \langle u\rangle),\\ (x_1,\ rec,\ \langle v\rangle)\end{array}\right\}} x_2, x_3, u, v \vdash L'(x_3, x_2) \mid L''(u, x_2, v).$$

The above production states that, once constraints on nodes x_1 and x_3 are satisfied by the environment, edge L is replaced by edges L' and L''. Edge L' has tentacles to nodes x_2 and x_3, while L'' is connected to x_3 and to two newly generated nodes u and v. Fusion substitution $[x_2/x_1]$ represents the mapping

$$\left\{\begin{array}{l} x_1 \mapsto x_2 \\ x_2 \mapsto x_2 \\ x_3 \mapsto x_3 \end{array}\right. \quad \textit{and determines the partition } \{\{x_1, x_2\}, \{x_3\}\},$$

where x_2 is the representative element of $\{x_1, x_2\}$.

The production can be graphically represented as follows:



This picture is similar to those in Section 4.1; however, here also new nodes are generated by the productions. Indeed, node u and v do not appear in the left-hand side of the production but are names of constraints in the production, hence they appear on the right-hand side. When this production synchronizes with other productions that offer conditions on x_1 and x_3 complementary to rec and send and exposes node v' and u' that will be fused with u and v after the sychronization.

Graphs over edge labels $\mathcal{L}$ and nodes $\mathcal{N}$ obey the usual structural congruence axioms in the same style of process calculi (e.g. π-calculus, KLAIM or the Ambient calculus). Moreover, recalling the informal description of hyperedge replacement described in Section 4.1, we can notice that nodes appearing in productions can be seen as "placeholders" that must be associated to nodes in a graph. Hence, we can freely rename nodes in productions; more precisely, given a production

$$x_1, ..., x_n \vdash L(x_1, ..., x_n) \xrightarrow[\pi]{\Lambda} \Gamma \vdash G,$$

renaming can be applied in several ways:

i. External nodes can be changed throughout the judgment, namely we can replace each x_i with a node y provided that $y \notin \{x_1, \ldots, x_n\} \cup \mathrm{fn}(G) \cup \mathrm{n}(\Lambda)$. For instance, if x_2 is replaced with y in production of Example 4.3.1, we obtain

$$x_1, y, x_3 \vdash L(x_1, y, x_3) \xrightarrow[[y/x_1]]{\left\{\begin{array}{c}(x_3,\ \overline{send},\ \langle u\rangle),\\ (x_1,\ rec,\ \langle v\rangle)\end{array}\right\}} y, x_3, u, v \vdash L'(x_3, y) \mid L''(u, y, v).$$

ii. nodes declared in $\mathrm{n}(\Lambda) - \Gamma$ can be α-converted. For instance, we can α-convert u (or v) in Example 4.3.1 with z and obtain the production

$$x_1, x_2, x_3 \vdash L(x_1, x_2, x_3) \xrightarrow[[x_2/x_1]]{\left\{\begin{array}{c}(x_3,\ \overline{send},\ \langle z\rangle),\\ (x_1,\ rec,\ \langle v\rangle)\end{array}\right\}} x_2, x_3, z, v \vdash L'(x_3, x_2) \mid L''(z, x_2, v).$$

iii. the representative nodes chosen by π can be consistently changed. For instance, in Example 4.3.1 nothing prevents us to use x_1 instead of x_2 as the representative element of the equivalence class $\{x_1, x_2\}$; in this case we would have the production

$$x_1, x_2, x_3 \vdash L(x_1, x_2, x_3) \xrightarrow[[x_1/x_2]]{\left\{\begin{array}{c}(x_3,\ \overline{send},\ \langle u\rangle),\\ (x_1,\ rec,\ \langle v\rangle)\end{array}\right\}} x_1, x_3, u, v \vdash L'(x_3, x_1) \mid L''(u, x_1, v).$$

As already stated, it is not mandatory that *all* edges take place in replacements, namely, some components can remain *idle* while others are replaced. *Idle productions* are defined to formalize this intuition:

Definition 4.3.2 (Idle productions) *Let id be the identity function on the set of nodes* $\{x_1, ..., x_n\}$.

An identity productions *is a production of the form*

$$x_1, ..., x_n \vdash L(x_1, ..., x_n) \xrightarrow[id]{\emptyset} x_1, ..., x_n \vdash L(x_1, ..., x_n).$$

An idle production *is either an identity production or a production of the form*

$$x_1, ..., x_n \vdash nil \xrightarrow[id]{\emptyset} x_1, ..., x_n \vdash nil$$

4.3.2 Transitions of graphs

Productions (idle or not) are synchronized via the inference rules in Table 4.3. Graph semantics is based on productions to specify edge replacement, while inference rules essentially synchronize productions and confer dynamic behaviour to graphs.

A transition is a logical judgment

$$\Gamma_1 \vdash G_1 \xrightarrow[\pi]{\Lambda} \gg \Gamma_2 \vdash G_2 \tag{4.4}$$

where Λ, π, Γ_2 and G_2 obeys the same conditions imposed on productions. Essentially, transitions can be seen as productions having general graphs on their left hand side. Hence transitions describe the dynamic evolutions of graphs.

Transition (4.4) states that $\Gamma_1 \vdash G_1$ can take part to rewritings that match constraints Λ and determine fusion substitution π. Once such conditions are satisfied, $\Gamma_1 \vdash G_1$ rewrites as $\Gamma_2 \vdash G_2$.

Definition 4.3.3 (Graph transitions) *Let $\langle \Gamma_0 \vdash G_0, \mathcal{P} \rangle$ be a graph rewriting system. The set of* transitions *$T(\mathcal{P})$ is the smallest set that contains $\mathcal{P}$, any idle production and that is closed under the four inference rules in Table 4.3.*

A derivation is obtained by starting from the initial graph and by executing a sequence of transitions, each obtained by synchronizing productions. The synchronization of rewriting rules requires matching of the actions and unification of the third components of the constraints Λ. After productions are applied, the unification function is used to obtain the final graph by merging the corresponding nodes.

In Table 4.3 we use notation $[^{v_1,\ldots,v_n}/_{u_1,\ldots,u_n}]$ (abbreviated as $[^{\vec{v}}/_{\vec{u}}]$) to denote substitutions that are applied both to graphs and sets of constraints. If $\rho = [^{\vec{v}}/_{\vec{u}}]$ is a substitution then ρG is the graph obtained by substituting all free occurrences of u_i with v_i in G for each $i = 1, ..., n$, while $\rho\Lambda = \{(x, a, \rho\vec{y}) : (x, a, \vec{y}) \in \Lambda\}$ where $\rho\vec{y}$ is the sequence $\rho(y_1), \ldots, \rho(y_h)$ if $\vec{y} = y_1, \ldots, y_h$.

Finally, given a function $f : A \to B$ and $y \in A$, $f_{-y} : A \setminus y \to B$ is defined as $f_{-y}(x) = f(x)$, for all $x \in A \setminus y$.

The most important rules in Table 4.3 are $(merge1)$ and $(merge2)$. They regulates how nodes can be fused. Rule $(merge1)$ fuses two nodes provided that no constraint is required on one of them, whereas rule $(merge2)$ handles with nodes upon which complementary actions are required. Rule (res) describes how graph transitions can be performed under node restriction. Finally, rule (par) states how transitions on disconnected graphs can be combined together.

Let us comment more on all the rules.

Rule $(merge1)$ fuses nodes x and y provided that no constraint is imposed on y, i.e. $\Lambda(y) \uparrow$. Premise $x \simeq_\pi y \Rightarrow \pi(y) \neq y$ imposes that, in case y is fused with a node x such that $x \simeq_\pi y$ (namely x and y are in the same equivalence class) then y must not be the representative element. However, if $x \not\simeq_\pi y$ fusion $[^x/_y]$ is possible; indeed, condition $x \simeq_\pi y \Rightarrow \pi(y) \neq y$ trivially holds.

$$(merge1)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma' \vdash G' \qquad \Lambda(y) \uparrow \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad \rho = [{}^{\pi(x)}/_{\pi(y)}]}{\Gamma \vdash [{}^{x}/_{y}]G \xrightarrow[(\pi;\rho)_{-y}]{\rho\Lambda}\!\!\gg \mathrm{n}(\rho\Lambda) \cup (\pi;\rho)_{-y}(\Gamma) \vdash \rho G'}$$

$$(merge2)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda \cup \{(x,a,\vec{v}),(y,\overline{a},\vec{w})\}}\!\!\gg \Gamma' \vdash G' \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad \rho = mgu\{[{}^{[x/_y]\vec{w}}/_{[x/_y]\vec{v}}], [{}^{\pi(x)}/_{\pi(y)}]\} \qquad \Gamma'' = \mathrm{n}(\rho\Lambda) \cup (\pi;\rho)_{-y}(\Gamma) \qquad U = \rho(\Gamma') \setminus \Gamma''}{\Gamma \vdash [{}^{x}/_{y}]G \xrightarrow[(\pi;\rho)_{-y}]{(\rho\Lambda \cup (x,\tau,\langle\rangle))}\!\!\gg \Gamma'' \vdash \nu\, U.\rho G'}$$

$$(res)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma' \vdash G' \qquad \Lambda(y) \uparrow \vee \Lambda(y) = (\tau, \langle\rangle) \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad U = \Gamma' \setminus (\mathrm{n}(\Lambda) \cup \pi_{-y}(\Gamma))}{\Gamma \vdash \nu\, y.G \xrightarrow[\pi_{-y}]{\Lambda \setminus (y,\tau,\langle\rangle)}\!\!\gg \mathrm{n}(\Lambda) \cup \pi_{-y}(\Gamma) \vdash \nu\, U.G'}$$

$$(par)\quad \frac{\Gamma_1 \vdash G_1 \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma_2 \vdash G_2 \qquad \Gamma_1' \vdash G_1' \xrightarrow[\pi']{\Lambda'}\!\!\gg \Gamma_2' \vdash G_2' \qquad (\Gamma_1 \cup \Gamma_2) \cap (\Gamma_1' \cup \Gamma_2') = \emptyset}{\Gamma_1 \cup \Gamma_1' \vdash G_1 \mid G_1' \xrightarrow[\pi \cup \pi']{\Lambda \cup \Lambda'}\!\!\gg \Gamma_2 \cup \Gamma_2' \vdash G_2 \mid G_2'}$$

Table 4.3: Inference rules for graph synchronization

A transition from $\Gamma, y \vdash G$ may be re-formulated to obtain the transition where y and x are coalesced, provided that fusion of their representative elements, ρ, is reflected on Λ, on π and on continuation $\Gamma' \vdash G'$. Indeed, if y is fused with x, also the other nodes equivalent to them are fused; the fusion substitution in the conclusion of $(merge1)$ is $\pi;\rho$ (restricted to Γ), all occurrences of $\pi(y)$ are replaced with $\pi(x)$ in $\mathrm{n}(\Lambda)$ and the final graph is $\rho G'$. It is obtained by merging $\pi(y)$ and $\pi(x)$ in G'.

Rule $(merge2)$ synchronizes complementary actions. The rule permits merging x and y in a transition where they offer complementary non-silent actions. As for $(merge1)$, x cannot replace the representative element of its equivalence class. Most general unifier ρ takes into account possible equalities due to the transitive closure of substitutions $[^{\vec{v}}/_{\vec{u}}]$ after $[^{x}/_{y}]$ has been applied. ρ fuses the corresponding nodes of the constraints and propagates previous fusions π. The resulting constraints $\rho\Lambda \cup \{(x, \tau, \langle\rangle)\}$ does not change constraints offered on nodes different from x and y (up to the necessary fusion ρ). Fusion substitution $(\pi;\rho)_{-y}$ acts on Γ by applying ρ. Finally, nodes U are the restricted nodes of $\rho G'$ and are those nodes that neither are in $(\pi;\rho)_{-y}(\Gamma)$ nor are generated by $\rho\Lambda$. This corresponds to the *close* rule of π-calculus.

Observation 4.3.2 *We remark that node x mentioned in rules $(merge1)$ and $(merge2)$ is a node in Γ appearing in the rules. This immediately follows for the condition $x \simeq_\pi y \Rightarrow y \neq \pi(y)$.*

Rule (res) deals with node restriction. Representative elements cannot be restricted if other nodes are in their equivalence class. Furthermore, only nodes can be restricted where either a synchronization action takes place or no constraint is imposed. If those conditions hold, the (possible) silent action on y is hidden and nodes not in $\Gamma' \setminus (\mathrm{n}(\Lambda) \cup \pi_{-y}(\Gamma))$ are restricted.

Rule (par) simply combines together disconnected judgments. Function $\pi \cup \pi'$ applied to a node x is $\pi(x)$ or $\pi'(x)$ depending on $x \in \Gamma_1$ or $x \in \Gamma'_1$. Condition $(\Gamma_1 \cup \Gamma_2) \cap (\Gamma'_1 \cup \Gamma'_2) = \emptyset$; guarantees that $\pi \cup \pi'$ is well defined because it implies $\Gamma_1 \cap \Gamma'_1 = \emptyset$, moreover such condition states that

- graphs $\Gamma_1 \vdash G_1$ and $\Gamma_2 \vdash G_2$ are disjoint;
- names generated by the transition of $\Gamma_i \vdash G_i$ (for $i \in \{1, 2\}$) do not occur neither as names of the other graph nor as names generated by its transition.

Rules in Table 4.3 guarantee that *idle transitions* can be derived for any well-formed graph, as stated by the following theorem:

Theorem 4.3.1 *For any well-formed graph $\Gamma \vdash G$, the (idle) transition*

$$\Gamma \vdash G \xrightarrow[id]{\emptyset} \Gamma \vdash G$$

can be derived from rules in Table 4.3.

PROOF. The proof is given by induction on the structure of G. If $G = nil$ then the thesis follows by the assumption that idle productions are always available.
If $G = L(y_1, ..., y_n)$ where L is an edge in $\mathcal{L}_{u \to v}$ such that $n = u + v$ and y_i's are nodes in Γ. Then, by hypothesis,

$$x_1, ..., x_n \vdash L(x_1, ..., x_n) \xrightarrow[id]{\emptyset} x_1, ..., x_n \vdash L(x_1, ..., x_n)$$

is an available identity production, hence, by applying rule $(merge1)$, we can fuse nodes x_i's in order to restrict to a judgment that contains only nodes y_j's.
Let us now assume that $G = G_1 \mid G_2$, then, by inductive hypothesis, the following transitions can be derived

$$\Gamma \vdash G_1 \xrightarrow[id]{\emptyset} \Gamma \vdash G_1$$

$$\Gamma \vdash G_2 \xrightarrow[id]{\emptyset} \Gamma \vdash G_2.$$

Nodes in Γ appearing in the last transition can be uniformly α-renamed, hence can derive the following transition:

$$\sigma\Gamma \vdash \sigma G_2 \xrightarrow[id']{\emptyset} \sigma\Gamma \vdash \sigma G_2$$

where σ is an injective substitution such that $\Gamma \cap \sigma\Gamma = \emptyset$ and id' is the identity function on $\sigma\Gamma$. We can now apply the (par) rule and obtain the transition

$$\Gamma \cup \sigma\Gamma \vdash G_1 \mid \sigma G_2 \xrightarrow[id \cup id']{\emptyset} \Gamma \cup \sigma\Gamma \vdash G_1 \mid \sigma G_2.$$

Finally, we can repeatedly apply rule $(merge1)$ to fuse nodes $\sigma\Gamma$ on the corresponding nodes in Γ, according to σ^{-1}.
The last case is $G = \nu\ x.G'$. By inductive hypothesis, $\Gamma, x \vdash G' \xrightarrow[id']{\emptyset} \Gamma, x \vdash G'$ (where id' is the identity on Γ, x) is a derivable transition and we can apply rule (res) to obtain the thesis. □

Despite of their small number, inference rules given in Table 4.3 seem quite involved (at least at a first glance). However we point out that designing a system using hypergraphs can take great advantage if one has in mind the intuitive description of hyperedge replacement given in Section 4.1. Indeed, in this case, the design reduces to the following step:

- representing the system as a number of components opportunely connected;
- for each component, the designer draws a hyperedge so that its external nodes are explicited (as in the reported example);

- then, on the external nodes, the designer writes the actions that will constraint the hyperedge rewriting;
- finally, the graph that will be substituted with the hyperedge is specified taking care that the external nodes of the graph satisfy the conditions of Definition 4.3.1.

4.4 Multiple Synchronizations

In [76] a different set of inference rules for graphs semantics has been presented. The main difference between the semantics proposed in [76] and rules in Table 4.3 lies in rules for merging nodes. In the former proposal any (finite) number of components can synchronize in a "single shot" provided complementary actions take place on different nodes. This section briefly report the semantics presented in [76] and discusses its peculiarities with respect to the new semantic rules in Section 4.3.

Definition 4.4.1 (Inference rules) *Let $\langle \Gamma \vdash G_0, \mathcal{P} \rangle$ be a hypergraph rewriting system and let $\Gamma \vdash G_1 \stackrel{\Lambda,\pi}{\rightarrow} \phi \vdash G_2$ be a transition where $\Delta = n(\Lambda) - \Gamma$ and $\phi = \pi(\Gamma) \cup \Delta$; The set $T'(\mathcal{P})$ of transitions is obtained from the productions $\mathcal{P}$ using the inference rules in Table 4.4 where the side conditions of the rules are:*

$$\psi_1 \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \begin{cases} \Delta \cap \sigma(\Gamma) = \emptyset \text{ and } \forall x \in \Delta . \sigma(x) = x \\ \sigma(x) = \sigma(y) \wedge \Lambda(x) \downarrow \wedge \Lambda(y) \downarrow \wedge x \neq y \Rightarrow \\ \quad (\forall z \notin \{x, y\} . \sigma(z) = \sigma(x) \Rightarrow \Lambda(z) \uparrow) \\ \quad \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (\overline{a}, \vec{w}) \wedge a \neq \tau \\ \rho = mgu(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (\overline{a}, \vec{w})\} \\ \qquad\qquad \cup \{\pi(x) = \pi(y) | \sigma(x) = \sigma(y)\}) \\ \Lambda'(z) = \begin{cases} (\tau, \langle\rangle), & \text{if } \sigma(x) = \sigma(y) = z \wedge x \neq y \wedge \Lambda(x) \downarrow \wedge \Lambda(y) \downarrow \\ \rho(\sigma(\Lambda))(z), & \text{otherwise} \end{cases} \\ \pi'(\sigma(x)) = \sigma(\rho(\pi(x))) \\ \vec{u} = \rho(\sigma(\phi)) - \phi' \end{cases}$$

$$\psi_2 \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \begin{cases} (\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x \\ \Lambda(x) \uparrow \text{ or } \Lambda(x) = (\tau, \langle\rangle), \\ \Lambda' = \Lambda - (x, \tau, \langle\rangle) \\ \vec{z} = \phi - \phi' \end{cases}$$

Rules (par') and (res') behaves exactly as the corresponding (par) and (res) rules of Table 4.3.

Rule ($merge$) is the rule for synchronization. The rule states that in a transition it is possible to merge any two nodes x and y that offer complementary non-silent actions (conditions on σ). The mapping ρ is the most general unifier that fuse the corresponding names of the actions and propagates the previous fusions (determined

$$(par') \ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2 \qquad \Gamma' \vdash G_1' \xrightarrow{\Lambda',\pi'} \phi' \vdash G_2'}{\Gamma, \Gamma' \vdash G_1 \mid G_1' \xrightarrow{\Lambda \cup \Lambda', \pi \cup \pi'} \phi, \phi' \vdash G_2 \mid G_2'} \quad \text{where } (\Gamma \cup \phi) \cap (\Gamma' \cup \phi') = \emptyset$$

$$(merge) \ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\sigma\Gamma \vdash \sigma G_1 \xrightarrow{\Lambda',\pi'} \phi' \vdash \nu \ \vec{u}.\rho(\sigma(G_2))} \quad \text{where } \psi_1 \text{ holds}$$

$$(res') \ \frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\Gamma \vdash \nu \ x.G_1 \xrightarrow{\Lambda',\pi|_\Gamma} \phi' \vdash \nu \ \vec{z}.G_2} \quad \text{where } \psi_2 \text{ holds}$$

Table 4.4: Inference rules for graph synchronization

by π). The label Λ' takes into account all possible synchronizations and leaves unchanged the actions offered on the other nodes up to the necessary fusions (ρ and σ). The new fusion substitution π' acts on $\sigma(\Gamma)$ by applying to it the mgu ρ. Finally, the names in ϕ after the fusion which are not present in $\phi' = \pi'(\Gamma) \cup (n(\Lambda') - \sigma(\Gamma))$ are restricted; this corresponds to the close rule of the π-calculus.

Intuitively, rules ($merge1$) and ($merge2$) in Table 4.3 are both encompassed by rule ($merge$) which can contemporary deal with idle nodes or with nodes where complementary actions take place. Moreover, it should be evident that the multiple synchronizations that can be derived with rule ($merge$) are also derivable by successive applications of ($merge1$) and ($merge2$). Hence the "expressive" power of the two transition systems is equivalent.

Even if much more complex than the semantic rules in Table 4.3, rules in Table 4.4 have some technical benefits. First, it is possible to describe concurrent behaviour of separate parts of a system "in one shot". Second, the proofs derived from this transition system require less steps than the inference rules of 4.3, hence with those rules it is also possible to prove some later results (theorems in Sections 5.4.1, 6.3.6 and in Section 6.3.6).

Chapter 5

A Hypergraphs-based semantics for Ambients

Abstract

The Ambient calculus is one of the best studied models addressing the needs of global computing, and it has acquired the rôle of touchstone for the most recent proposals. However the interactive, abstract semantics of ambients is still not fully explored. In fact, as it is the case of most foundational calculi for global computing, reduction semantics for ambients has been found to be simpler than the corresponding labelled transition system (LTS) semantics. However, reduction semantics has the main disadvantage with respect to LTS semantics that it makes harder to define, and reason about, abstract compositional behavior.

We show how it is possible to translate the fragment of the Ambient calculus introduced in Section 3.2 in our graph synchronization framework maintaining the semantics of processes. The translation also underlines how Ambient primitives require to synchronize more components at once (processes and ambients) in order to model the desired behaviour.

Contents

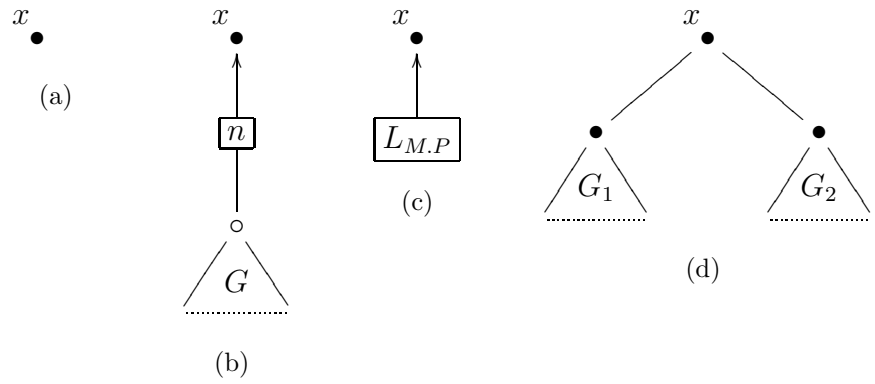


Figure 5.1: Representing the translation function

5.1 A Graphical Ambient Calculus

This section introduce a translation function that maps terms of the Ambient calculus into the graphical calculus presented in Chapter 4.

Definition 5.1.1 (From Ambient to graphs)

$$
\begin{array}{rcll}
[\![\, \mathbf{0} \,]\!]_x & = & x \vdash nil & \\
[\![\, n[P] \,]\!]_x & = & x \vdash \nu\, y.(G \mid n(y,x)), & \text{if } y \neq x \wedge [\![\, P \,]\!]_y = y \vdash G \\
[\![\, M.P \,]\!]_x & = & x \vdash L_{M.P}(x) & \\
[\![\, P_1 | P_2 \,]\!]_x & = & x \vdash G_1 \mid G_2, & \text{if } [\![\, P_i \,]\!]_x = x \vdash G_i, \text{where } i = 1,2 \\
[\![\, rec\, X.\, P \,]\!]_x & = & [\![\, P[^{rec\, X.\, P}/_X] \,]\!]_x &
\end{array}
$$

Definition 5.1.1 introduces the mapping function $[\![\, P \,]\!]_x$ that returns a graph whose only free node x corresponds to the root of the ambient process P.

In the above translation, sequential processes $M.P$ are directly represented by edges labelled by $L_{M.P}$. We say that $L_{M.P}(x)$ is an *incoming edge* in x. While this requires an infinite number of edge labels, it is easy to see that only a finite number of them (and of the corresponding activity rules defined below) is needed to derive all computations of any particular ambient.

The graph associated to the **0** process is an isolated node. The graph of $n[P]$ with free node x is obtained by constructing G, the graph of P, on node y, attaching it to the graph $n(y,x)$ and restricting y. Conventionally, we say that $n(x,y)$ is an *outgoing* edge from x and an *incoming* edge in y. Note that the ambient name n is interpreted as an edge from y to x labelled n. Ambient names N and sequential processes are the only edge labels. The parallel composition $P_1 \mid P_2$ is obtained by making the graph of P_1 and P_2 to share their root node x. Figure 5.1 graphically represents the translation function in the case described so far; arrowed tentacles remind the incoming edges in x. Finally, recursive processes are unfolded first[1].

[1]Note that the $[\![\, \_ \,]\!]_x$ is well defined because recursion variables are guarded by capabilities.

Definition 5.1.1 enjoys many properties that can be useful later. We collect some lemmas that are used to prove the main results of this section.

Lemma 5.1.1 *For any ambient process P,* $\mathrm{fn}([\![P]\!]_x) = \{x\}$.

Proof. Note that x appear as interface node in all rule of Definition 5.1.1, hence $x \in \mathrm{fn}([\![P]\!]_x)$. By construction, we use always new nodes for translating sub-terms of P and any node different from x is restricted. We can therefore conclude that $\mathrm{fn}([\![P]\!]_x) = \{x\}$. □

Lemma 5.1.2 $[\![\_]\!]_\_$ *preserves structural congruence, namely, for any ambient processes P and Q such that $P \equiv Q$, $[\![P]\!]_x$ and $[\![Q]\!]_x$ are structural equivalent graphs.*

Proof. The proof easily descends by induction on the structure of P and Q. We detail only the most difficult case. If $P \equiv Q_1 \mid Q_2$ for some Q_1 and Q_2, then

- either $Q_2 = \mathbf{0}$ and $P \equiv Q_1$
- or there are P_1 and P_2 ($\neq \mathbf{0}$) such that $P = P_1 \mid P_2$ and $P_i \equiv Q_i$, $i = 1, 2$.

(the symmetric cases are analogous and can be threated similarly).
In the former case, by definition, $[\![Q_1 \mid Q_2]\!]_x = [\![Q_1]\!]_x \otimes x \vdash nil = x \vdash G \mid nil$. By structural congruence of hypergraphs and inductive hypothesis, we conclude that $x \vdash G \mid nil = x \vdash G = [\![P]\!]_x$.
In the latter case, $[\![P_i]\!]_x \equiv [\![Q_i]\!]_x$, $i = 1, 2$ by inductive hypothesis and, by definition of $\otimes$ and structural congruence of hypergraphs, we obtain the thesis. □

The given translation is injective but not surjective. However, the graphs $[\![P]\!]_x$ in the image of the translation function can be characterized in terms of *ambient graphs*.

Definition 5.1.2 (Ambient graphs) *Given a graph $\Gamma \vdash G$, a node x in G is a* root *node whether no edge in G is an outgoing edge from x.*

An ambient graph *is a graph labelled on $LE = \{L_{M.P} | M.P \in Proc\} \cup N$ which*

1. *is acyclic;*
2. *every node has at most one outgoing edge labelled in N;*
3. *there is only one free node and it is the* root *node.*

Proposition 5.1.1 *If P is an ambient process, then $[\![P]\!]_x$ is an ambient graph.*

PROOF. The proof is given by structural induction on P. The base cases are $[\![\ \mathbf{0}\ ]\!]_x$ and $[\![\ M.P\ ]\!]_x$ that, by construction, are ambient graph.

Let us consider the ambient process, $n[P]$ case and assume that $[\![\ n[P]\ ]\!]_x = x \vdash (\nu\ y).(G \mid n(y,x))$, where $[\![\ P\ ]\!]_y = y \vdash G$ under the assumption $y \neq x$. Since $y \neq x$ and, by inductive hypothesis, y is the unique root node in $y \vdash G$, then x is the unique root node in $[\![\ n[P]\ ]\!]_x$. Furthermore, y has one outgoing edge, namely $n(y,x)$, in $[\![\ n[P]\ ]\!]_x$. Finally, $[\![\ n[P]\ ]\!]_x$ is acyclic because, by inductive hypothesis, $[\![\ P\ ]\!]_y$ is acyclic and if, *by absurdum*, we assume that $G \mid n(y,x) \equiv (\nu\ X)(L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k)) \mid G'$ for some nodes X, some edges L_i's and a term G' and $L_1(\vec{x}_1) \mid ... \mid L_k(\vec{x}_k)$ is a cycle, then

- either $n(y,x)$ does not appear in the cycle,
- or there is a $L_i(\vec{x}_i)$ that is equal to $n(y,x)$.

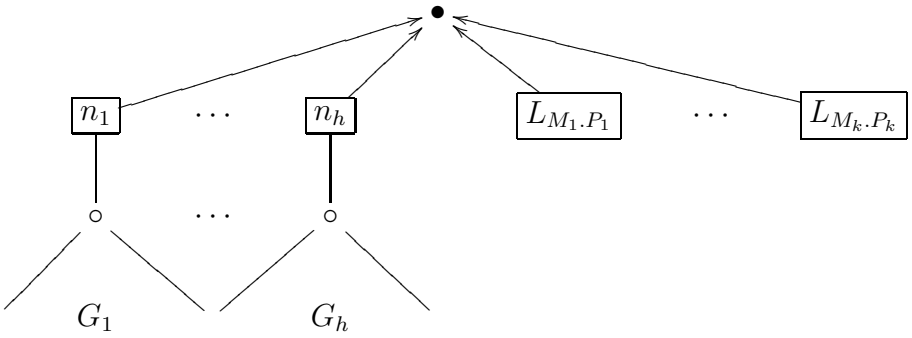
In the former case, we obtain a contradiction because the cycle would also be a cycle in $[\![\ P\ ]\!]_y$. In the latter case, since x is a root node, only the last edge of the path can be $n(y,x)$, whereas all the other edges are edges of G. Therefore, by inductive hypothesis, $L_1(\vec{x}_1) \mid ... \mid L_{k-1}(\vec{x}_{k-1})$ is not a cycle, hence L_1 should be an outgoing edge from x which contradicts the hypothesis that $x \notin \mathrm{n}(G)$.

The remaining cases are similarly proved. □

Theorem 5.1.1 *$[\![\ \_\ ]\!]_{\_}$ is a bijection on ambient graphs (up to structural equivalence).*

PROOF. Lemma 5.1.2 and Proposition 5.1.1 imply that $[\![\ \_\ ]\!]_{\_}$ is an injection from the equivalence classes induced by structural congruence on the set of ambient processes to the equivalence classes induced on hypergraphs by their structural congruence relation.

In order to prove that it also is surjective on ambient graphs, we can simply consider that the general shape of an ambient graph is



that corresponds to the ambient process $n_1[P_1] \mid ... \mid n_h[P_h] \mid L_{M_1.P_1} \mid ... \mid L_{M_k.P_k}$ where, for $i = 1, ..., h$, P_i is the ambient process that corresponds to ambient graph G_i. □

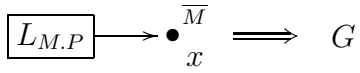
5.2 Productions for Ambient

We now define the productions of our version of the Ambient calculus. There are two kinds of productions: *activity productions*, relative to sequential processes, and *coordination productions* that edges (corresponding to ambients) perform for orchestrating the activity of ambient processes. Intuitively, activity productions correspond to ambient capability, while coordination productions are those productions that ambient edges must fire in order to permit activity productions to produce the desired effect.

Definition 5.2.1 (Activity productions) *The activity productions have the following form.*

$$x \vdash L_{M.P}(x) \xrightarrow{\{(x, \overline{M}, \langle\rangle)\}} \llbracket\ P\ \rrbracket_x \tag{5.1}$$

where $\llbracket\ P\ \rrbracket_x = x \vdash G$.
A graphical representation of transition (5.1) is



where, in general, when $(x, \mu, \langle y \rangle) \in \Gamma$, *node* x *in the right member is labelled by* x, μ.

Activity productions determine the actions that sequential processes are able to perform. In our approach, sequential processes become edge labels: when an action is performed, an edge labelled by $L_{M.P}$ is rewritten as the graph corresponding to P.

The complementary actions to synchronize the activity productions must be offered by ambients; more precisely, ambients must signal their existence emitting the complementary actions on their attaching nodes and, in this manner, performing the correct synchronized steps.

Definition 5.2.2 (Coordination productions) Coordination productions *are detailed below. For every production, we give both the sequent and its graphical representation. In the latter, the left-hand-side and the right-hand-side of a production are drawn in the style of Definition 5.1.1.*
Open coordination production

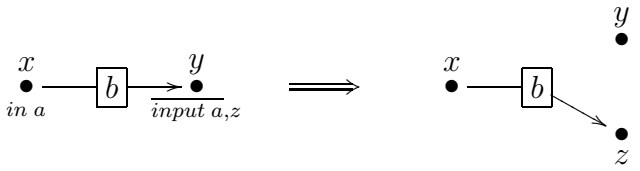
$$x, y \vdash a(x, y) \xrightarrow[[y/x]]{\{(y, open\ a, \langle\rangle)\}} y \vdash nil$$

(*open*)

Input coordination productions

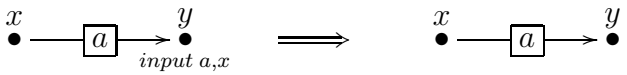
$$x, y \vdash b(x, y) \xrightarrow{\{(x, in\ a, \langle\rangle), (y, \overline{input\ a}, \langle z\rangle)\}} x, y, z \vdash b(x, z)$$

($input1$)



$$x, y \vdash a(x, y) \xrightarrow{\{(y, input\ a, \langle x\rangle)\}} x, y \vdash a(x, y)$$

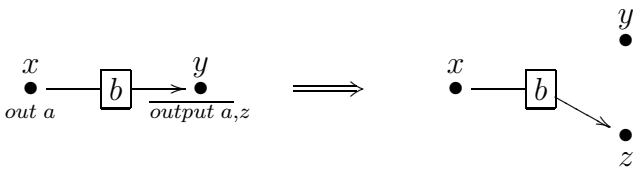
($input2$)



Output coordination productions

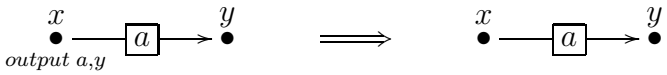
$$x, y \vdash b(x, y) \xrightarrow{\{(x, out\ a, \langle\rangle), (y, \overline{output\ a}, \langle z\rangle)\}} x, y, z \vdash b(x, z)$$

($output1$)



$$x, y \vdash a(x, y) \xrightarrow{\{(x, output\ a, \langle y\rangle)\}} x, y \vdash a(x, y)$$

($output2$)



Coordination productions define the complementary actions that ambients must perform in order to synchronize themselves with edges representing sequential processes (that declare the activity productions).

The (*open*) production states that if the ambient a has a parallel process that wants to open it, then the edge corresponding to a disappears and x is fused with y. Observe how this mechanism permits any edge connected at x "to move" on y; moreover, this is completely transparent to edges connected at y.

Remark 5.2.1 *We point out that fusion substitutions constitute the distinguished feature of our graphical calculus with respect to [97]. They play an important rôle for*

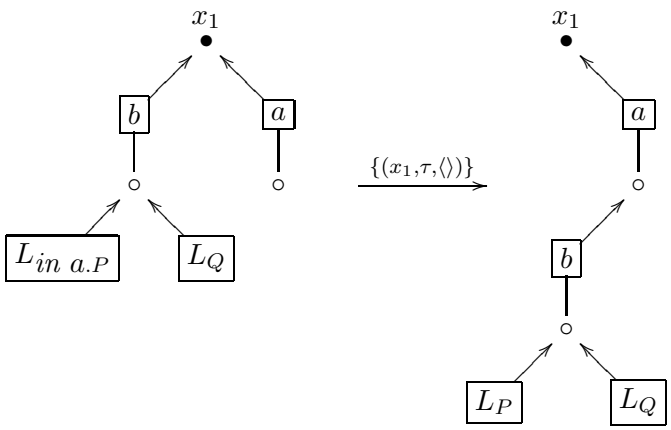


Figure 5.2: Graph transition

faithfully representing the semantics of the open capability of Ambient. Indeed, the synchronized hyperedge replacement mechanism proposed in [97] cannot represent the open capability in a simple way because two existing nodes cannot be coalesced.

Coordinating *in* actions requires two productions:

- Production (*input*1) asserts that, if a process inside b wants to drive b in an ambient a, then the destination of b will become the new node z.
- On the other hand, production (*input*2) controls the entrance of an external process in a: this production simply passes the source x of a to the entering process.

Analogously to the input productions, (*output*1) and (*output*2) take care of the output action. We remark that (*output*1) acts quite similarly to (*input*1).

5.3 An Extended Example

As an example we show the correspondence between an Ambient calculus reduction and the corresponding graph transition. Let us consider the ambient reduction

$$b[in\ a.P \mid Q] \mid a[\mathbf{0}] \rightarrow a[b[P \mid Q]]$$

where P and Q are sequential processes.

Following Definition 5.1.1 and hypergraph semantics (see Definition 4.3.3, page 76) we should obtain the transition of Figure 5.2 where the left-hand-side of the transition represents the ambient process $[\![\ b[in\ a.P \mid Q] \mid a[\mathbf{0}]\ ]\!]_{x_1}$, whereas the right-hand-side is the graph corresponding to $[\![\ a[b[P \mid Q]]\ ]\!]_{x_1}$.

We apply the inference rules of Table 4.3 in order to construct a proof for transition in Figure 5.2.

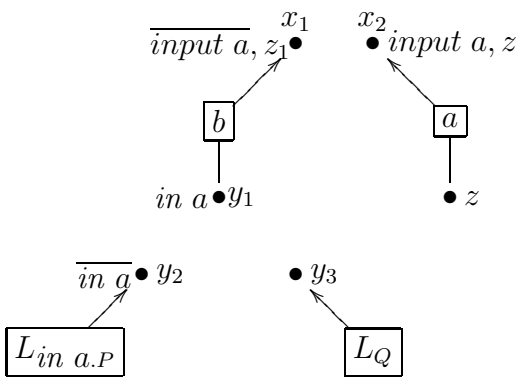


Figure 5.3: Graph decomposition

First (step 1) we decompose the graph in its elementary edges and determine the productions that correspond to the elementary components of the transition.

$$x_1, y_1 \vdash b(y_1, x_1) \xrightarrow[id]{\left\{\begin{array}{l}(y_1, in\ a, \langle\rangle),\\ (x_1, \overline{input\ a}, \langle z_1\rangle)\end{array}\right\}} x_1, y_1, z_1 \vdash b(y_1, z_1) \tag{5.2}$$

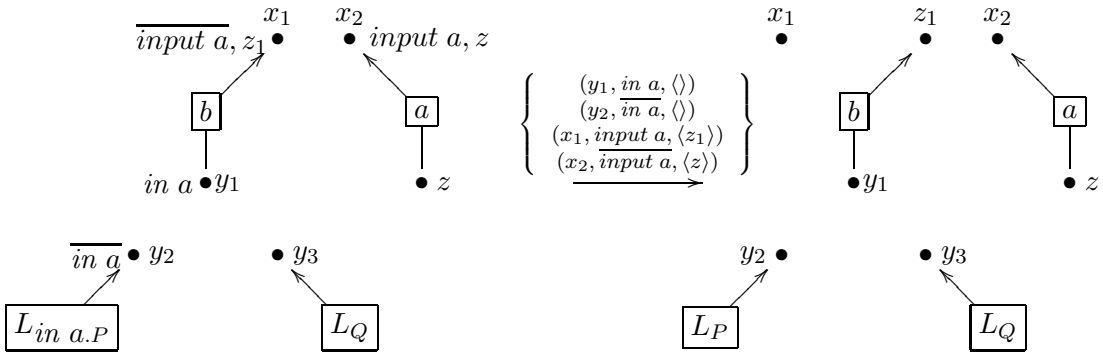
$$x_2, z \vdash a(z, x_2) \xrightarrow[id]{\{(x_2, input\ a, \langle z_1\rangle)\}} x_2, z \vdash a(z, x_2) \tag{5.3}$$

$$y_2 \vdash L_{in\ a.P}(y_2) \xrightarrow[id]{\{(y_2, \overline{in\ a}, \langle\rangle)\}} y_2 \vdash L_P(y_2) \tag{5.4}$$

$$y_3 \vdash L_Q(y_3) \xrightarrow[id]{\emptyset} y_3 \vdash L_Q(y_3) \tag{5.5}$$

Transitions (5.2) and (5.3) are instances of the coordination productions (*input*1) and (*input*2), respectively; transition (5.4) is the activity production relative to *in* $a.P$ and transition (5.5) is the identity transition that leaves L_Q idle.

The graphical representation is given in Figure 5.3. By applying the (*par*) rule to the productions (5.2), (5.3), (5.4) and (5.5) we obtain the graph below:



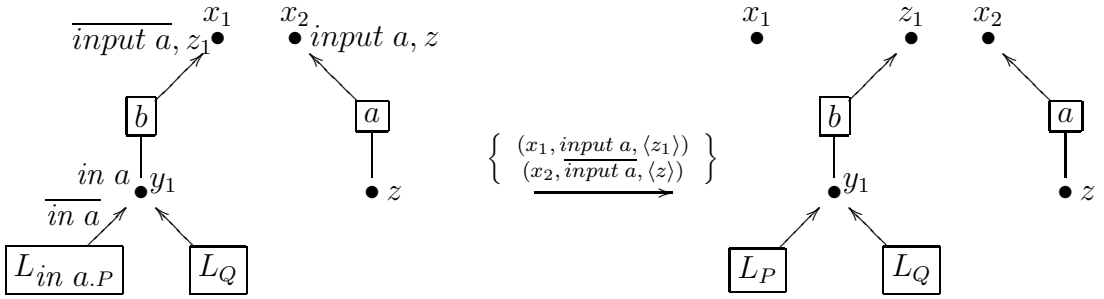


Figure 5.4: A part of the proof

In terms of sequents we have the transition

$$\Gamma \vdash G_1 \xrightarrow[id]{\left\{\begin{array}{l}(x_1, \overline{input\ a}, \langle z_1 \rangle),\\ (x_2, input\ a, \langle z_1 \rangle)\\ (y_1, \overline{in\ a}, \langle\rangle)\\ (y_2, \overline{in\ a}, \langle\rangle)\end{array}\right\}} \gg \Gamma, z_1 \vdash G_2 \tag{5.6}$$

where

$$\begin{array}{lcl} G_1 & = & b(y_1, x_1) \mid a(z, x_2) \mid L_{in\ a.P}(y_2) \mid L_Q(y_3) \\ G_2 & = & b(y_1, z_1) \mid a(z, x_2) \mid L_P(y_2) \mid L_Q(y_3) \\ \Gamma & = & \{x_1, x_2, y_1, y_2, y_3, z\}. \end{array}$$

The application of the merge rules (step 2) provides the fusion of the nodes in order to obtain a graph of the same shape of the ambient process but without restricted nodes. In Λ there are two pairs of complementary actions over x_1, x_2 and y_1, y_2. Let us first consider action on y_1 and y_2, we can apply rule $(merge2)$ with the substitution $\sigma = [^{y_2}/_{y_1}]$ fuses y_2 onto y_1. Substitution σ determines $\rho = \sigma$ and $U = \{z_1\}$. We, therefore, obtain the transition

$$\Gamma \setminus y_2 \vdash \sigma G_1 \xrightarrow[id]{\Lambda_1} \gg \{x_1, x_2, y_1, y_3, z, z_1\} \vdash b(y_1, z_1) \mid a(z, x_2) \mid L_P(y_1) \mid L_Q(y_3) \tag{5.7}$$

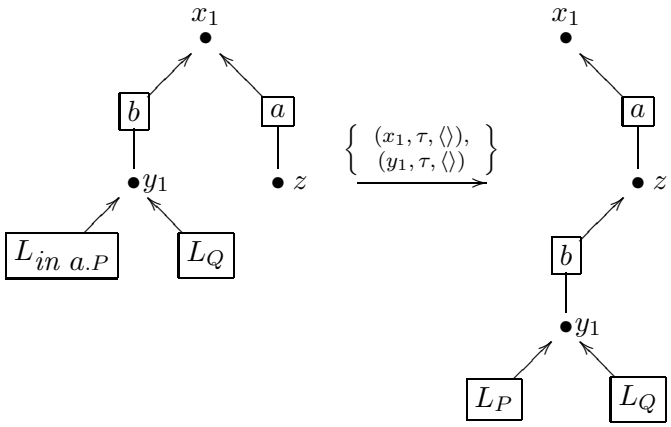
where $\Lambda_1 = \{(x_1, \overline{input\ a}, \langle z_1 \rangle), (x_2, input\ a, \langle z \rangle), (y_1, \tau, \langle\rangle)\}$. Production 5.7 allows us to apply $(merge1)$ and to fuse y_1 and y_3 which yields:

$$\Gamma \setminus \{y_2, y_3\} \vdash \sigma[^{y_1}/_{y_3}]G_1 \xrightarrow[id]{\Lambda_1} \gg \{x_1, x_2, y_1, z, z_1\} \vdash b(y_1, z_1) \mid a(z, x_2) \mid L_P(y_1) \mid L_Q(y_1).$$

Note that $\rho = \sigma$ and, since $y \notin n(\Lambda)$, $\rho\Lambda = \Lambda$. The graphical representation of the so far computed proof is given in Figure 5.4. By observing that Λ' has two complementary actions, we can again apply $(merge2)$ with the fusion $\sigma' = [^{x_1}/_{x_2}]$. Differently from 5.7, $\rho = [^{x_1,z}/_{x_2,z_1}]$ that is due to the fusion of name z_1, exported in the $\overline{input\ a}$ action with z. The resulting sequent is

$$x_1, y_1, z \vdash \begin{array}{c} b(y_1, x_1) \mid a(x_1, z) \mid \\ L_{in\ a.P}(y_1) \mid L_Q(y_1) \end{array} \xrightarrow[[^z/_{z_1}]]{\left\{\begin{array}{l}(x_1, \tau, \langle\rangle),\\ (y_1, \tau, \langle\rangle)\end{array}\right\}} x_1, y_1, z \vdash \begin{array}{c} b(y_1, z) \mid a(x_1, z) \mid \\ L_P(y_1) \mid L_Q(y_1) \end{array}$$

that is graphically represented as



We remark that the above transition requires a synchronization involving three edges and two nodes: the edges relative to *in* $a.P$ and b that synchronize on node y_1, and the edges relative to ambients b and a that synchronize on node x_1. This makes clear that the *in* capability of ambients requires to synchronize three components (the *out* capability is analogous).

Finally, two applications of the (res) rule (step 3) are needed in order to restrict nodes z and y_1. This concludes the proof of the transition.

5.4 Semantics Correspondence

The example presented in the previous section applies a general technique that can be exploited to show how reductions of Ambient terms can be mimicked by applying graph transition rules of Table 4.3 to productions for Ambient presented in Section 5.2.

The first result states that each Ambient reduction has a corresponding graph transition:

Theorem 5.4.1 *For all ambient process* $P \in Proc$, *if* $P \rightarrow Q$ *then* $[\![P]\!]_x \xrightarrow[id]{\Lambda} [\![Q]\!]_x$ *and either* $\Lambda = \emptyset$ *or* $\Lambda = \{(x, \tau, \langle\rangle)\}$.

Proof. We proceed by induction on the length of the proof of $P \rightarrow Q$.
The base cases are the axioms of Table 3.7. Let us first consider the axion for open prefix that we state below:

$$open\ n.P_1 \mid n[P_2] \rightarrow P_1 \mid P_2. \tag{5.8}$$

The graph corresponding to the left-hand-side of reduction (5.8) is

$$x \vdash L_{open\ n.P_1}(x) \mid \nu\ y.(G \mid n(y, x)) \tag{5.9}$$

where G is such that $[\![P_2]\!]_y = y \vdash G$. We proceed, as in Section 5.3, by (i) decomposing graph (5.9) into its constituent part, (ii) by synchronizing them and, (iii) by properly restricting nodes.

First, we remark that we can derive transition $y \vdash G \xrightarrow[[y/y]]{\emptyset} y \vdash G$ (by Theorem 4.3.1). Let us consider the arc $n(y', x')$, where $x \neq x'$ and $y \neq x'$; by definition, we have the production $y', x' \vdash n(y', x') \xrightarrow[[x'/y']]{\{(x', open\ n, \langle\rangle)\}} x' \vdash nil$. We can apply rules (par) and $(merge1)$ to attach graph (5.9) on node and arc $n(y', x')$ by fusing y' and y.

$$x', y', y \vdash G \mid n(y', x') \xrightarrow[[y/y']]{\{(x', open\ n, \langle\rangle)\}} x', y \vdash G \mid n(y, x') \tag{5.10}$$

Notice that the application of $(merge1)$ is possible because we have an idle transition for $y \vdash G$, hence, y' can be substituted for y in G. The right-hand-side of transition (5.10) corresponds to applying the fusion substitution $[y/y']$ determined by rule $(merge1)$.

Let us consider the activity production for *open* that, by definition is:

$$x \vdash L_{open\ n.P_1}(x) \xrightarrow[[x/x]]{\{(x,\ \overline{open\ n},\ \langle\rangle)\}} [\![P_1]\!]_x,$$

we can again apply the (par) rule to it and to transition (5.10), obtaining the transition below:

$$x, x', y', y \vdash G \mid n(y', x') \mid L_{open\ n.P_1}(x) \xrightarrow[[y/y']]{\left\{ \begin{array}{c} (x,\ \overline{open\ n},\ \langle\rangle), \\ (x', open\ n, \langle\rangle) \end{array} \right\}} x, x', y \vdash G \mid n(y, x') \mid G', \tag{5.11}$$

where G' is such that $[\![P_1]\!]_x = x \vdash G'$. Then we apply rule $(merge2)$ for synchronizing the complementary actions in (5.11) which yields the transition:

$$x, x', y', y \vdash G \mid n(y', x) \mid L_{open\ n.P_1}(x) \xrightarrow[[y/y'][x/y]]{\{(x,\ \tau,\ \langle\rangle)\}} x \vdash [x/y]G \mid G'$$

Notice that step (iii) (i.e. restricting nodes) is not required, because the only candidate y' has been substituted for x in the final graph.

We omit the proof for the remaining axioms because they are dealt similarly to the *open* reductions. In particular, Section 5.3 shows how to handle reductions of *in* actions.

Now we consider the inductive step. If the Ambient reduction is obtained with a proof whose length is greater that one, then the last rule applied is one of

$$\frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \to Q}{n[P] \to n[Q]}.$$

By inductive hypothesis, we have that $[\![P]\!]_x \xrightarrow[id]{\Lambda\gg} [\![Q]\!]_x$, where Λ is either empty or is the singleton $\{(s,\ \tau,\ \langle\rangle)\}$. Then, by Theorem 4.3.1, we have that we can derive an idle transition for $[\![R]\!]_{x'}$ $(x \neq x')$, hence we can use rules (par) and $(merge1)$ to put the graphs in parallel and fuse nodes x' and x obtaining the thesis.
If the last rule applied is the rule for a reduction under an ambient n, we proceed as before. However, after having connected the graph for $[\![P]\!]_y$ to the ambient edge $n(y,x)$, we must restrict y. The application of (res) rule causes the (eventual) silent action in Λ to be removed. □

In general, a graph obtained by translating an ambient process can perform transitions where the target graphs do not correspond to any Ambient term. The reason is that graph semantics requires more steps for mimicking ambient reduction. However, if we restrict our attention to Ambient graphs and *basic transitions*, defined below, we can state Theorem 5.4.2 which, in some sense, is the inverse of Theorem 5.4.1.

Definition 5.4.1 (Basic transition) *A transition* $\Gamma \vdash G \xrightarrow[\pi]{\Lambda\gg} \Gamma' \vdash G'$ *is* basic *if, and only if,*

- π *is the identity function on* Γ*;*
- *there is a proof of* $\Gamma \vdash G \xrightarrow[\pi]{\Lambda\gg} \Gamma' \vdash G'$ *which uses exactly one instance of either* $(open)$ *or* $(input1)$ *or* $(output1)$*;*
- Λ *is either the singleton* $\{(x, \tau, \langle\rangle)\}$ *or it is empty.*

Theorem 5.4.2 *Let* P *be a term in Proc. If* $[\![P]\!]_x \xrightarrow[\pi]{\Lambda\gg} \Gamma \vdash G$ *is a basic transition, then either* $[\![P]\!]_x = \Gamma \vdash G$ *or there is a process* $Q \in Proc$ *such that* $\Gamma \vdash G = [\![Q]\!]_x$ *(up to structural equivalence) and* $P \to Q$*.*

Proof. First we observe that, by definition, idle productions are basic, hence if $[\![P]\!]_x \xrightarrow[\pi]{\Lambda\gg} \Gamma \vdash G$ is an idle production, then $\Gamma \vdash G = [\![P]\!]_x$ hence the proof is trivially proved. Triviality of the previous argument authorizes us to no longer consider idle productions in the rest of the proof.

Hereafter, the proof proceeds by induction on the structure of P. The basic case is $P \equiv M.P$. The only transitions that can be derived from $[\![P]\!]_x$ are activity of coordination productions (that are not basic) or idle productions, hence the statement of the theorem trivially holds.

Let P be the ambient term $n[Q]$; by definition $[\![P]\!]_x = \nu\ y.(G|n(y,x))$, where $[\![Q]\!]_y = y \vdash G$ and $y \neq x$. Hence, we can discard all the proofs where coordination or activity productions of $n(y,x)$ are exploited because they imposes constraints

on x that cannot be removed neither with rule (res) nor with synchronization on x. The only possibility for deriving a basic transition is from transitions of the inner graph $y \vdash G$. Reasoning as before, we can avoid considering productions that synchronize G and $n(y,x)$ because, by inspecting productions of $n(y,x)$ this would impose constraints on x. The only possibility is that $y \vdash G$ has a basic transition, say $y \vdash G \xrightarrow[\pi]{\Lambda} y \vdash G'$. Since $[\![Q]\!]_y = y \vdash G$, by inductive hypothesis we have that $Q \rightarrow Q'$ and $[\![Q']\!]_y = y \vdash G'$. Moreover, Ambient semantics allows us to derive the transition $n[Q] \rightarrow n[Q']$ (see Section 3.2.2, page 49). The above discussion should easily convince that the only (kind of) proof that keeps edge $n(y,x)$ idle and "lifts" the basic transition of G to $[\![P]\!]_x$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{y \vdash G \xrightarrow[\pi]{\Lambda} y \vdash G'} \qquad x,z \vdash n(z,x) \xrightarrow[[x,z/x,z]]{\emptyset} x,z \vdash n(z,x)}{x,y,z \vdash G|n(z,x) \xrightarrow[[x,y,z/x,y,z]]{\Lambda} x,y,z \vdash G'|n(z,x)}\,(par)}{x,y \vdash G|n(y,x) \xrightarrow[[x,y/x,y]]{\Lambda} x,y \vdash G'|n(y,x)}\,(merge1)}{x \vdash \nu\, y.(G|n(y,x)) \xrightarrow[[x/x]]{\Lambda'} x \vdash \nu\, y.(G'|n(y,x))}\,(res)} \tag{5.12}$$

Hence, by definition $[\![n[Q]]\!]_x = x \vdash \nu\ y.(G'|n(y,x))$ which concludes the proof in this case.

Let us assume that $P \equiv P_1|P_2$; by definition $[\![P]\!]_x = x \vdash G_1|G_2$ where $x \vdash G_i = [\![P_i]\!]_x$ $(i = 1,2)$. The basic transitions that can be inferred for $[\![P]\!]_x$ are either the basic transitions of P_1 (or of P_2) or the transition obtained by synchronizing P_1 and P_2. The former case has a proof completely analogous to the proof of the previous case. We consider now the latter case. If we have a basic transition whose proof contains an instance of $(open)$ production that synchronize P_1 and P_2 then one of P_1, P_2 is an ambient process and the other is a process prefixed by action $open\, a$. Let us assume $P_1 = a[Q_1]$ and $P_2 = open\ a.Q_2$. Hence, by Ambient semantics, we have that $P_1|P_2 \rightarrow Q_1|Q_2$ that corresponds to the synchronization of the corresponding graphs that provides the basic transition. The other case are dealt similarly.

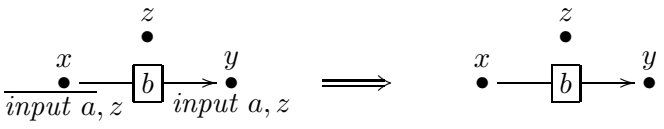
The last case is when P is a recursive call; the proof easily follows by the fact that the translation of P is the unfolding of the recursive call and the inductive hypothesis. □

5.5 Remote actions

In order to respect the standard semantics of Ambient calculus, productions for *in a* prefix require moving ambients to have a sibling ambient called a. This "neighborhood" condition reflects the reduction rule of the ambient calculus. However, we can relax such requirement and consider the possibility of having "remote" *in a* in the sense that the moving ambient can enter an ambient a that is a sub-ambient of one of its sibling ambients.

The idea is to add a production (*input3*) that may forward *input* signals to its inner ambients

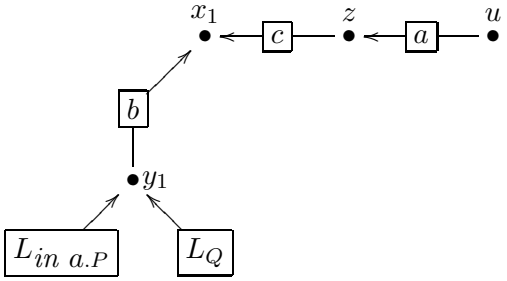
(*input3*)



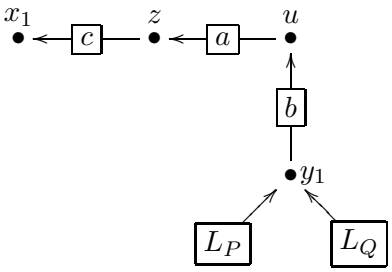
$$x, y \vdash b(x, y) \xrightarrow{\{(x, \overline{input\ a}, \langle z\rangle), (y, input\ a, \langle z_1\rangle)\}} x, y, z \vdash b(x, y).$$

Production (*input3*) says that if edge $b(x, y)$ receives a request on y for entering an ambient a, then it propagates the signal to its inner processes (together with the attaching node provided by the pilot process that issued the $\overline{in}$ signal). If the signal will reach an edge $a(\_, \_)$ and such edge will execute with production *input2* then the remote *in* transition will finish by opportunely connecting edge $a(\_, \_)$ and the migrating graph (as for the non-remote *in* actions).

In order to give an intuition of how the remote *in* works, consider the following graph:



Using production *input3* we may obtain the following graph:



Finally, we remark that coordination and activity productions for *out* actions are very similar. Therefore, similarly to remote *in* , remote out actions can be states as well.

Chapter 6

Application Level QoS

Abstract

Modern applications aim at exploiting WAN for purposes that can be considered "ambitious" if compared to the initial motivation of their deployment. For instance, multimedia applications are becoming very popular over the Internet. Programmers would enjoy new paradigm for specifying the "minimal level" of bandwidth, reliability, security, etc. that the communication infrastructure should guarantee in order to have "reasonable" execution environment. A formal framework for stating and reasoning on these applications is desirable as well.

This chapter presents QLAIM that introduces some mechanisms for specifying and reasoning with Quality of Services (QoS) attribute in WAN applications. QLAIM is translated in the graphical calculus of Chapter 4; the translation models routing with path reservation.

Contents

6.1 Specifying Application Level Quality of Services

Wide-Area Network (WAN) applications have become one of the most important class of applications in distributed computing. Currently, Internet and World Wide Web are the primary environments for designing, developing and distributing applications. Network services have now evolved into self-contained components which inter-operate easily with each other by supporting WEB-based access protocols [102]. In addition, network services may adapt themselves to match the particular capabilities of a variety of devices ranging from traditional PCs, to Personal Digital Assistants and Mobile Phones having intermittent connectivity to the network.

In this new scenario both final users and WAN application designers put special emphasis on Quality of Service (QoS) issues. Here, QoS is a measure of properties of applications such as security, performance, bandwidth, transaction support, reliability, working cost, etc. In general, QoS attributes are special parameters of network services. For final users, the perceived QoS of their computations is not only given by the performance of WEB servers but also by the availability of certain resources and by the flow of network traffic. *Awareness* of these information is crucial for choosing the best network services that match user's requirements. For instance, final users can react to phenomena like network congestion by binding their network devices to different sites where the requested services are available. Similarly, QoS awareness is exploited by WAN application designers to control resource usages and resource accesses in order to ensure and maintain certain security levels and to provide users with differentiated QoS's.

Distributed middlewares and programming languages permit applications to control network connectivity and resource accesses. A paradigmatic example is provided by the Java programming language through the SOCKET and the SECURITY API's. However, existing middlewares and API do not allow a direct control over QoS attributes. Moreover, most research on QoS is *system oriented* in the sense that it has its focus on properties of the lower layers of the Internet protocol stack.

We believe that QoS awareness at *application* level, i.e. *application-oriented QoS*, is a key requirement for most of WAN applications and is expected to become the *added-value* capability of the emerging evolutionary WAN applications.

At a foundational level, several process calculi have been developed to gain a more precise understanding of mobility and security. In particular, some approaches addressed the problem of resources access control for mobile processes [59, 93, 95, 30, 42]. These works led to the introduction of suitable type systems where types specify and fully characterize access control policies. However, a foundational model for QoS awareness is still missing. Some preliminary results in this direction can be found in [35]. Notably, Cardelli and Davies introduce a calculus which incorporates a notion of communication rate (bandwidth) and describe some programming constructs based on this calculus.

Our research goal is to contribute at providing a formal understanding of application-oriented QoS, as a step toward the development of proof techniques and tools for the automated verification and certification of properties of WAN applications. We introduce a declarative approach to the specification of QoS attributes at application level. Here, declarative means that QoS attributes are related to the abstractions application programmers deal with. Hence, QoS attributes are used to select and configure the underlying system-oriented QoS mechanisms.

6.2 KLAIM and QoS

We have abstracted the basic features of the problem by extending KLAIM with primitives for specifying network connections together with QoS attributes over them. The main characteristics of KLAIM are code mobility and *network awareness*, namely, the fact that programmers can explicitly refer localities such that they can control resource and computation distribution. We consider a variant of KLAIM [19] that provides mechanisms to program dynamic changes of network connectivity. Hence, peer groups take the form of *graphs* with dynamically evolving edges.

Albeit localities are first class citizens in KLAIM, they are used at a very high level of abstraction. For instance, a process allocated on a site s can access the tuple space at s' by simply naming s' in a remote action. In a more realistic setting, this is not always the case: s and s' are connected by a path of (eventually) intermediate sites and an edge of this path can be (momentarily) out of order or can be as congested as being considered not viable.

In order to overcome this modeling issues, KLAIM has been extended with features that allow explicit declaration of links connecting sites of a net; interaction between two sites need that a path of those links connects the sites [19]. One of our contribution is that we further extend the calculus presented in [19] with the definition of a calculus for application-oriented QoS. To this purpose, KLAIM networking constructs [19] are extended with attributes which are used to specify the QoS properties of network links. This QoS attributes can be thought of as being simply a value specifying the *abstract* cost of using a given link.

Example 6.2.1 *An instance of abstract cost can be a pair $\langle c, \delta \rangle$ that specifies the set δ of* access rights *(in the sense of [59]) of using the link. More generally, an abstract cost can be a vector whose components represent different aspects of network connectivity such as bandwidth, working cost, latency, length, reliability, and security.*

Links are parameterized with respect to a set of values, ranged over by κ. Such values are used to specify the features of a network link. We also assume existence of a binary operation $\oplus$ that permits composing cost values. Of course, the definition of the operation $\oplus$ depends on the form chosen for κ.

Nets:	N	::=	...	*As for* KLAIM
		|	$s ::^I O\mathbb{P}$	*Single site*
Coordinators:	$\mathbb{P}$	::=	$\mathbf{0}$	*Empty process*
		|	$\mathbf{out}(t)$	*Tuple*
		|	$\gamma.\mathbb{P}$	*Action prefixing*
		|	$\mathbb{P}_1 \mid \mathbb{P}_2$	*Parallel composition*
		|	$\mathbb{A}\langle\widetilde{\ell},\widetilde{e}\rangle$	*Coordinator Invocation*
Coordinator operations:	γ	::=	...	KLAIM *action (without* **new***)*
		|	$\mathbf{new}(u,\mathbb{P})$	*Site creation*
		|	$\mathbf{login}(\ell,\kappa)$	*Connection request*
		|	$\mathbf{accept}(u,\kappa)$	*Connection acceptance*
		|	$\mathbf{logout}(\ell,\kappa)$	*Dis-connection (by client)*
		|	$\mathbf{disc}(()\ell,\kappa)$	*Dis-connection (by server)*

Table 6.1: QLAIM Syntax

Example 6.2.2 *If we consider $\kappa = \langle c, \delta\rangle$ as done in Example 6.2.1, then we could define:*

$$\langle c_1, \delta_1\rangle \oplus \langle c_2, \delta_2\rangle = \langle c_1 + c_2, \delta_1 \cap \delta_2\rangle$$

We will refer to this extension of KLAIM as QLAIM. Next sections presents syntax and semantics of QLAIM.

6.2.1 QLAIM syntax

The syntax of QLAIM dialect is a slight variation of the syntax of KLAIM introduced in 3.3.1. The calculus introduces the concept of *coordinator processes*, ranged over by $\mathbb{P}$, that can be thought of as a network operating system residing on the sites of the net. Conversely, KLAIM processes can be thought of as the user programs that may invoke system calls in the site. We will call *standard processes* the KLAIM processes that do not contain **new** actions: Indeed, we will allow only coordinator processes to create new sites. Standard processes will still be ranged over by P.

Coordinators are special processes that cannot migrate (indeed, they cannot be used as an arguments of **eval**). They are installed at a site either when the site is initially configured or when the site is dynamically created by performing $\mathbf{new}(u,\mathbb{P})$. As usual, we assume that each coordinator identifier $\mathbb{A}$ has an associated defining equation $\mathbb{A}(\tilde{u},\tilde{x}) \stackrel{\text{def}}{=} \mathbb{P}$.

Table 6.1 lists productions for QLAIM. As for KLAIM, a net is the parallel composition of sites. However, differently from QLAIM, a site is equipped with two new components: I is the set of links *entering in* s, and O is the set of links *exiting from* s. Elements from I and O are pairs of the form $\langle s, \kappa\rangle$, where s is a site and κ is a cost. For I to be well formed, we require that if $\langle s_1, \kappa_1\rangle$ and $\langle s_2, \kappa_2\rangle$ belong to I, then it must either be $s_1 \neq s_2$ or $\kappa_1 \neq \kappa_2$. For a net N to be well-formed we

require that for any pair of its sites $s_1 ::^{I_1} O_1\mathbb{P}_1$ and $s_2 ::^{I_2} O_2\mathbb{P}_2$, if $\langle s_2, \kappa\rangle \in O_1$ (for any κ) then $\langle s_1, \kappa\rangle \in I_2$.

Given $s ::^{I} O\mathbb{P}$, if $\langle s', \kappa\rangle \in I$ ($\langle s', \kappa\rangle \in O$ respectively) then κ specifies the features of the the link connecting s' to s (s to s', respectively). We also say that s is a *gateway* for sites that occur in I, while sites that occur in O are gateways for s.

Observation 6.2.1 *In general, a site can have more than one gateway that have different costs.* QLAIM *permits more connection between two sites too because sets I and O can contain pairs $\langle s, \kappa\rangle$ or $\langle s, \kappa'\rangle$.*

Gateways may be thought of as being the nodes of a path connecting two sites, therefore, if s_2 is a gateway for s_1 and s_3 is a gateway for s_2, then s_3 is a gateway for s_1 too. Gateways are essential for processes to be able to perform remote operations because a remote operation can be performed only if there exists a path of gateways from the site where it runs to the target site of the operation. Any QLAIM site plays a twofold rôle: *i*) It is a computational environment hosting processes and tuple spaces and *ii*) a site act as *gateway* for other sites of the net. Moreover, sites can act both as clients (belonging to a specific subnet) and as servers (taking in charge of, possibly private, subnets).

Coordinator processes can perform five new action in addition to the standard process KLAIM operations. These new actions are $\mathbf{new}(u, \mathbb{P})$, $\mathbf{login}(\ell, \kappa)$, $\mathbf{accept}(u, \kappa)$, $\mathbf{logout}(\ell, \kappa)$ and $\mathbf{disc}(()\ell, \kappa)$; all of them are oriented to model dynamic changes of the network topology (creation of new sites, creation and removal of links). Notice that all these operations are not indexed with a locality, since they always act locally at the site where they are executed. Note that coordination processes can also be the parallel composition of many processes.

6.2.2 QLAIM semantics

The operational semantics of the language is defined by a standard labelled transition system. The semantics is given in terms of a transition relation, $\stackrel{\kappa}{\succ\!\!\longrightarrow}$, that describes possible net evolutions and the relative abstract cost κ (when κ is missing, the operation is local). Relation $\stackrel{\kappa}{\succ\!\!\longrightarrow}$ relies on

- a labelled transition $\stackrel{a}{\underset{s,\kappa}{\longrightarrow\!\!\!\!\!\rightarrow}}$, that describes both process intentions to perform specific operations and the availability of resources (tuples and sites) in the net. Site s indicates the (last) gateway that made the operation possible, κ represents the cost of the operation and label a represents either a tuple, or a site, or the intended operation. Table 6.2 below collects the inference rules for the transition $\stackrel{a}{\underset{s,\kappa}{\longrightarrow\!\!\!\!\!\rightarrow}}$,

- a labelled relation $\stackrel{a}{\longrightarrow\!\!\!\!\!\rightarrow}$, that accounts for the intention of coordinators to perform one coordination operation. Label a is of the form $\mathbf{a}(u_1, arg, u_2)$,

where **a** denotes the operation, u_1 is the site performing it, u_2 is the target site and arg is the argument of **a**. For instance, $\mathbf{r}(s_1, T, s_2)$ represents $\mathbf{read}(T)@s_2$ performed from s_1. Rules defining $\stackrel{a}{\longrightarrow\!\!\!\!\!\rightarrow}$ are given in Table 6.3.

We now briefly comment on the rules in Tables 6.2, 6.3 and 6.4.

Axioms (TUPLE), (SITE), (EVAL), (IN) and (READ) use the execution site s of the process as first gateway with cost $\top$ that is the null cost. More formally, we assume that $\kappa \oplus \top = \kappa = \top \oplus \kappa$.

Rules (TUPLE) and (SITE) respectively signal the presence of tuple t or site $s ::^{IO} \mathbb{P}$ in the net. Rules (EVAL) says that process $\mathbf{eval}(P)@s'.\mathbb{P}$ running at s is willing to spawn process P for execution at s'. Notice that P is a standard process. Similarly, (IN) and (READ) says that input read prefixes of processes located at s are willing to access the tuple space located at s' (for removing and reading template T, respectively). Rule (ENV) implements the mechanism we have sketched before: When performing a remote operation **a** that uses s_1 as (last) gateway, a gateway s_2 for s_1 can be used provided that s_2 can act as gateway for s_1. Notice that only transitions with labels of the form $\mathbf{a}(-,-,-)$ can use a site different from the one performing the operation as a gateway. Last tree rules of Table 6.2 are analogous to the rules for KLAIM in Table 3.11 and in Table 3.12. In Table 6.2 we have omitted symmetric rules of (PRCOMP) (NETCOMP) and (ENV).

Table 6.3 specifies the transition rules for coordinator prefixes listed in Table 6.1.
Axiom (LOGIN) says that $\mathbf{login}(s_2, \kappa)$ logs the executing site s_1 in s_2 whose features are specified by κ. The effect of the transition is to add the pair $\langle s, \kappa \rangle$ to the outgoing gateways set O of s.
Axiom (LOGOUT) says that $\mathbf{logout}(s_2, \kappa)$ disconnects the gateway κ connecting s_1 and s_2; as a consequence, $\langle s_2, - \rangle$ is removed from the O-component of site s_1.
Action **accept** is the complementary action of **login** and rule (ACCEPT) says that, for a $\mathbf{login}(s_1, \kappa')$ executed at s_2 to succeed, at s_1 there must be a coordinator of the form $\mathbf{accept}(u, \kappa).\mathbb{P}'$. The premise $\kappa \models \kappa'$ gives an early semantics flavour to the rule: Intuitively, it means that **accept** authorizes the establishment of a link with features given by κ' that do not exceed the specified cost κ. As a consequence of this synchronization, $\langle s_1, \kappa \rangle$ is added to the I-component of sites s_2.
Rule (DISCONNECT) is similar to (LOGOUT), but in this case the disconnection is required by the site acting as gateway and involve the I-component of site s.
Rule (NEWLOC) says that $\mathbf{new}(u, \mathbb{P})$ aims at creating a new site, s_2, in the net and binds it to u. Process $\mathbb{P}$ is the coordinator of the newly created site. Site s_2 can be considered as a "private" site that can be accessed by other sites only if s_1 communicates the value of variable u, which is the only way to access the fresh site. Predicate $s_2 \notin s_1 ::^{IO} \mathbb{P} \mid \mathbb{P}'$ means that: $s_2 \neq s_1$, $\langle s_2, \kappa \rangle \notin I \cup O$ for any κ or l, and s_2 does not syntactically occurs in $\mathbb{P}$ and $\mathbb{P}'$ (predicate $s \notin N_1 \parallel N_2$ has a similar meaning). Notice that a **new** prefix does not automatically logs the new site in the generating one. This can be done by installing a coordinator in the new site that performs a **login**.

$$s ::^{IO} \mathbf{out}(t) \xrightarrow[s,\top]{t@s} s ::^{IO} \mathbf{0} \qquad \text{(Tuple)}$$

$$s ::^{IO} \mathbb{P} \xrightarrow[s,\top]{s ::^{IO} \mathbb{P}} \mathbf{0} \qquad \text{(Site)}$$

$$s ::^{IO} \mathbf{eval}(P)@s'.\mathbb{P} \xrightarrow[s,\top]{\mathbf{e}(s,P,s')} s ::^{IO} \mathbb{P} \qquad \text{(Eval)}$$

$$s ::^{IO} \mathbf{in}(T)@s'.\mathbb{P} \xrightarrow[s,\top]{\mathbf{i}(s,T,s')} s ::^{IO} \mathbb{P} \qquad \text{(In)}$$

$$s ::^{IO} \mathbf{read}(T)@s'.\mathbb{P} \xrightarrow[s,\top]{\mathbf{r}(s,T,s')} s ::^{IO} \mathbb{P} \qquad \text{(Read)}$$

$$\frac{N_1 \xrightarrow[s_1,\kappa]{\mathbf{a}(s,-,s')} N_1' \qquad N_2 \xrightarrow[s_2,\top]{s_2 ::^{I,\langle s_1,\kappa_1\rangle O} \mathbb{P}} N_2'}{N_1 \parallel N_2 \xrightarrow[s_2,\kappa \otimes \kappa_1]{\mathbf{a}(s,-,s')} N_1' \parallel N_2' \parallel s_2 ::^{I,\langle s_1,\kappa_1\rangle O} \mathbb{P}} \qquad \text{(Env)}$$

$$\frac{s ::^{IO} \mathbb{P}_1 \xrightarrow[s,\kappa]{a} s ::^{IO} \mathbb{P}_1'}{s ::^{IO} \mathbb{P}_1|\mathbb{P}_2 \xrightarrow[s,\kappa]{a} s ::^{IO} \mathbb{P}_1'|\mathbb{P}_2} \ \mathrm{bn}(a) \cap \mathrm{fn}(\mathbb{P}_2) = \emptyset \qquad \text{(PrComp)}$$

$$\frac{s ::^{IO} \mathbb{P}[{}^{\widetilde{\ell},\widetilde{v}}/_{\widetilde{u},\widetilde{x}}] \xrightarrow[s,\kappa]{a} N}{s ::^{IO} A\langle \widetilde{\ell}, \widetilde{v}\rangle \xrightarrow[s,\kappa]{a} N} \ A(\widetilde{u},\widetilde{x}) \stackrel{def}{=} \mathbb{P} \qquad \text{(PrDef)}$$

$$\frac{N_1 \xrightarrow[s,\kappa]{a} N_1'}{N_1 \parallel N_2 \xrightarrow[s,\kappa]{a} N_1' \parallel N_2} \ \mathrm{bn}(a) \cap \mathrm{fn}(N_2) = \emptyset \qquad \text{(NetComp)}$$

Table 6.2: Process Semantics

$$s_1 ::^{IO} \mathbf{login}(s_2, \kappa).\mathbb{P} \xrightarrow{\mathbf{lin}(s_1, \kappa, s_2)} s_1 ::^{IO, \langle s_2, \kappa \rangle} \mathbb{P} \qquad \text{(Login)}$$

$$s_1 ::^{IO, \langle s_2, \kappa \rangle} \mathbf{logout}(s_2, \kappa).\mathbb{P} \xrightarrow{\mathbf{lout}(s_1, \kappa, s_2)} s_1 ::^{IO} \mathbb{P} \qquad \text{(Logout)}$$

$$\frac{\kappa \models \kappa'}{s_1 ::^{IO} \mathbf{accept}(u, \kappa).\mathbb{P} \xrightarrow{\mathbf{acc}(s_1, \kappa', s_2)} s_1 ::^{I, \langle s_2, \kappa' \rangle O} \mathbb{P}[{}^{s_2}/_{u}]} \qquad \text{(Accept)}$$

$$s_1 ::^{I, \langle s_2, \kappa \rangle O} \mathbf{disc}(()s_2, \kappa).\mathbb{P} \xrightarrow{\mathbf{dis}(s_1, \kappa, s_2)} s_1 ::^{IO} \mathbb{P} \qquad \text{(Disconnect)}$$

$$\frac{s_2 \notin s_1 ::^{IO} \mathbb{P} | \mathbb{P}'}{s_1 ::^{IO} \mathbf{new}(u, \mathbb{P}).\mathbb{P}' \xrightarrow{\mathbf{n}(s_1, \mathbb{P}, s_2)} s_1 ::^{IO} \mathbb{P}'[{}^{s_2}/_{u}]} \qquad \text{(Newloc)}$$

$$\frac{s_1 ::^{IO} \mathbb{P}_1 \xrightarrow{a} s_1 ::^{I'O'} \mathbb{P}'_1 \qquad a \neq \mathbf{n}(s_1, \mathbb{P}, s_2)}{s_1 ::^{IO} \mathbb{P}_1 | \mathbb{P}_2 \xrightarrow{a} s_1 ::^{I'O'} \mathbb{P}'_1 | \mathbb{P}_2} \qquad \text{(Comp1)}$$

$$\frac{N_1 \xrightarrow{a} N'_1 \qquad a \neq \mathbf{n}(s_1, \mathbb{P}, s_2)}{N_1 \parallel N_2 \xrightarrow{a} N'_1 \parallel N_2} \qquad \text{(Comp2)}$$

$$\frac{s_1 ::^{IO} \mathbb{P}_1 \xrightarrow{\mathbf{n}(s_1, \mathbb{P}, s_2)} s_1 ::^{IO} \mathbb{P}'_1 \qquad s_2 \notin \mathbb{P}_2}{s_1 ::^{IO} \mathbb{P}_1 | \mathbb{P}_2 \xrightarrow{\mathbf{n}(s_1, \mathbb{P}, s_2)} s_1 ::^{IO} \mathbb{P}'_1 | \mathbb{P}_2} \qquad \text{(CompNew1)}$$

$$\frac{N_1 \xrightarrow{\mathbf{n}(s_1, \mathbb{P}, s_2)} N'_1 \qquad s_2 \notin N_1 \parallel N_2}{N_1 \parallel N_2 \xrightarrow{\mathbf{n}(s_1, \mathbb{P}, s_2)} N'_1 \parallel N_2} \qquad \text{(CompNew2)}$$

Table 6.3: Coordinator Semantics

The rest of the rules in Table 6.3 account for process composition and net composition, and differentiate the case when the set of sites in the net does not change from the case when the set is increased (due to the creation of a new site). In particular, rules (CompNew1) and (CompNew2), ensure that the site s_2 of the new site is actually fresh. Rules (Comp1), (Comp2), (CompNew1) and (CompNew2) have symmetric counterparts that have been omitted.

In Table 6.4 rule for network evolution are given.
Rule (NC) synchronizes **login** and **accept** actions on sites s_1 and s_2, respectively. Note that s_1 and s_2 agree on the cost κ that is assigned to the common link.
When a coordinator process issues a **logout** action from site s_1 toward a remote site s_2, (NDC) removes the item corresponding to the link from the I-component of s_2. Remember that (Logout) symmetrically updates the O-component of s_1.
(NDS) is similar and updates the I-component of s_2 when a **disc**( )is performed at s_1.
Rule (NNew) is a simple lifting of rule (Newloc) to the network level. As stated above, the new site s_2 is added to the net. Initially, it has no gateways and only coordinator process $\mathbb{P}$ is running at s_2.
Rule (NEv) says that the **eval** operation can take place only if the target site is present. The condition $\kappa \models \mathbf{e}$ imposes cost κ enables the operation **eval**.

Observation 6.2.2 *The definition of $\kappa \models \mathbf{a}$ depends on the form chosen for values κ. For instance, if $\kappa = \langle c, \delta \rangle$, where δ is a set of access rights one for each kind of enabled operation, then we could define $\langle c, \delta \rangle \models \mathbf{a}$ if, and only if, $\mathbf{a} \in \delta$. Notice that, coordination operations are always local, have no assigned cost and are always enabled.*

Finally, rules (NIn) and (NRead) behaves in a similar way. An input (read) action from s_1 at s_2 requires that a tuple t matching template T is present at s_2; notice that κ must encompass action **i** (**r**). In the continuation net, formal variables of T are replaced with the corresponding valued of t. We remark that for the input prefix the continuation is $N_2[^t/_T]$ that, according to rule (tuple) cancels tuple t from the tuple space of s_2; while the continuation net for the (NRead) is $N_1'[^t/_T]$ that corresponds to the net where t is not removed from s_2.

The labelled transition system semantics has the main drawback that it does not control and does not properly permit reasoning about QoS requirements. In particular, given a net N the operational semantics determines all the possible evolutions for all possible costs. In other words, it does not provide any formal mechanisms to pick up automatically the evolution having the minimal cost: the evolution which ensures the optimal QoS. In next section, we map Qlaim on the graphical calculus presented in 4.2; the mapping ensures that remote actions will always be "routed" on the minimal cost path.

$$\frac{N_1 \xrightarrow{\mathbf{lin}(s_1,\kappa,s_2)} N_1' \qquad N_2 \xrightarrow{\mathbf{acc}(s_2,\kappa,s_1)} N_2'}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2'} \quad \text{(NC)}$$

$$\frac{N_1 \xrightarrow{\mathbf{lout}(s_1,\kappa,s_2)} N_1' \qquad N_2 \xrightarrow[s_2,\top]{s_2 ::^{I,\langle s_1,\kappa\rangle O} \mathbb{P}} N_2'}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2' \parallel s_2 ::^{IO} \mathbb{P}} \quad \text{(NDC)}$$

$$\frac{N_1 \xrightarrow{\mathbf{dis}(s_1,\kappa,s_2)} N_1' \qquad N_2 \xrightarrow[s_2,\top]{s_2 ::^{IO,\langle s_1,\kappa\rangle} \mathbb{P}} N_2'}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2' \parallel s_2 ::^{IO} \mathbb{P}} \quad \text{(NDS)}$$

$$\frac{N_1 \xrightarrow{\mathbf{n}(s_1,\mathbb{P},s_2)} N_2}{N_1 \rightarrowtail N_2 \parallel s_2 ::^{\emptyset\emptyset} \mathbb{P}} \quad \text{(NNew)}$$

$$\frac{N_1 \xrightarrow[s_2,\kappa]{\mathbf{e}(s_1,P,s_2)} N_1' \qquad N_1' \xrightarrow[s_2,\top]{s_2 ::^{IO} \mathbb{P}} N_2 \qquad \kappa \models \mathbf{e}}{N_1 \overset{\kappa}{\rightarrowtail} N_2 \parallel s_2 ::^{IO} P|\mathbb{P}} \quad \text{(NEv)}$$

$$\frac{N_1 \xrightarrow[s_2,\kappa]{t@s_2} N_1' \qquad N_1' \xrightarrow[s_2,\top]{\mathbf{i}(s_1,T,s_2)} N_2 \qquad match(T,t) \qquad \kappa \models \mathbf{i}}{N_1 \overset{\kappa}{\rightarrowtail} N_2[{}^{t}/_{T}]} \quad \text{(NIn)}$$

$$\frac{N_1 \xrightarrow[s_2,\kappa]{t@s_2} N_1' \qquad N_1' \xrightarrow[s_2,\top]{\mathbf{r}(s_1,T,s_2)} N_2 \qquad match(T,t) \qquad \kappa \models \mathbf{r}}{N_1 \overset{\kappa}{\rightarrowtail} N_1'[{}^{t}/_{T}]} \quad \text{(NRead)}$$

Table 6.4: Net Semantics

6.3 A Hypergraphs semantics for QLAIM

In this section, by exploiting the graphical calculus, we define an alternative semantics for QLAIM which takes care of QoS attributes. We first present a translation scheme from QLAIM nets and processes to the graphical calculus, then we present the productions that each hyperedge uses in the translation. In the rest of the chapter, we shall use the following notations. If $\Gamma_1 \vdash G_1$ and $\Gamma_2 \vdash G_2$ are two judgments, then $\Gamma_1 \vdash G_1 \otimes \Gamma_2 \vdash G_2$ denotes the judgment $\Gamma_1 \cup \Gamma_2 \vdash G_1 \mid G_2$.

6.3.1 QLAIM translation

The mapping function $[\![\_]\!]$ associates a hypergraph to a QLAIM net. We assume that processes appearing in the net are closed, i.e. all variable appearing in process terms are under the scope of a binder. It is defined by induction on the syntactical structure of QLAIM nets. The most important case is the translation of a QLAIM site: Let $s ::^I O\mathbb{P}$ be a site where $O = \{\langle s_1, \kappa_1\rangle, ..., \langle s_n, \kappa_n\rangle\}$ and $I = \{\langle s'_1, \kappa'_1\rangle, ..., \langle s'_m, \kappa'_m\rangle\}$ then we let $\Gamma = \{s, s_1, ..., s_n, s'_1, ..., s'_m\}$ and let $\vec{x}$ be a vector whose length is n and whose elements are pairwise distinct. Hereafter, we will write $\vec{x}$ in place of $x_1, ..., x_n$.

$$[\![s ::^I O\mathbb{P}]\!] \;=\; \Gamma \vdash (\nu\, \vec{x}, p, r)([\![\mathbb{P}]\!]_p \mid \Delta_n(\vec{x}, r) \mid \prod_{i=1}^{n} G_{s_i}^{\kappa_i}(x_i, s_i) \mid \mathfrak{S}_s(p, r, s)) \tag{6.1}$$

The hypergraph associated to $s ::^I O\mathbb{P}$ contains an edge $\mathfrak{S}_s(p, r, s)$ representing site s. The remaining nodes, p and r, are used for synchronizing $\mathfrak{S}_s$ with local processes (that are connected to p) and the router edge of s (that is connected to r). The hypergraph representing process $\mathbb{P}$ allocated at s is connected to $\mathfrak{S}_s$ on the node p, while router edge of s, Δ_n, is connected to $\mathfrak{S}_s$ through node r. The hypergraph in (6.1) also contains a node x_i for each outgoing gateway in O. Those nodes are part of the interface of edge Δ_n and are used to connect gateway edges $G_{s_i}^{\kappa_i}(x_i, s_i)$. A graphical representation is given in Figure 6.1. In some sense, site s is represented by edge $\mathfrak{S}_s(p, r, s)$ that interfaces incoming gateways, process executed at s and gateways departing from s. The dotted tentacles in Figure 6.1 aim at remarking that the hyperedges $G_s^{\kappa_j}$'s, corresponding to gateways entering in s, are not connected to nodes s'_j, but to a restricted node where routing edge of s'_j is connected.

Parallel composition of nets and empty net are trivially translated according to the following equations.

$$\begin{aligned} [\![N_1 \parallel N_2]\!] &= [\![N_1]\!] \otimes [\![N_2]\!] \\ [\![\mathbf{0}]\!] &= \emptyset \vdash nil \end{aligned}$$

Net $N_1 \parallel N_2$ is mapped to a hypergraph obtained by juxtaposing the hypergraphs of the constituent nets, N_1 and N_2; whereas, the empty net is mapped to the empty hypergraph, as formally defined below.

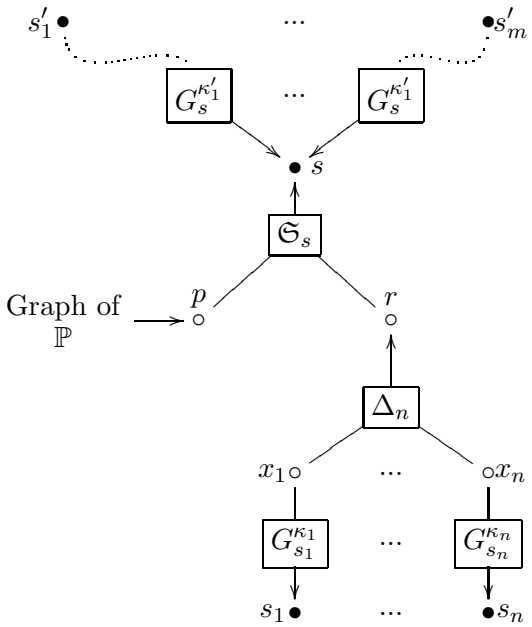


Figure 6.1: Graphs for QLAIM sites

The mapping for processes is described by the equations below:

$$
\begin{aligned}
[\![\ \mathbf{0}\ ]\!]_p &= nil \\
[\![\ \mathbf{out}(t)\ ]\!]_p &= L_{\mathbf{out}(t)}(p) \\
[\![\ \gamma.\mathbb{P}\ ]\!]_p &= L_{\gamma.\mathbb{P}}(p), \quad \text{if } \gamma \neq \mathbf{login}(s,\kappa) \\
[\![\ \mathbf{login}(s,\kappa).\mathbb{P}\ ]\!]_p &= \lambda_{\mathbf{login}(s,\kappa).\mathbb{P}}(s,p) \\
[\![\ \mathbb{P}_1 \mid \mathbb{P}_2\ ]\!]_p &= [\![\ \mathbb{P}_1\ ]\!]_p \mid [\![\ \mathbb{P}_2\ ]\!]_p \\
[\![\ \mathbb{A}\langle \widetilde{\ell}, \widetilde{v}\rangle\ ]\!]_p &= [\![\ \mathbb{P}[^{\widetilde{\ell},\widetilde{v}}/_{\widetilde{u},\widetilde{x}}]\ ]\!]_p, \quad \text{if } \mathbb{A}(\widetilde{u},\widetilde{x}) \stackrel{\text{def}}{=} \mathbb{P}.
\end{aligned}
$$

The hypergraph of a process $\mathbb{P}$ has an outgoing tentacle toward its execution site. The hypergraph relative to the empty process simply is the empty graph; tuple processes and non-**login** action prefixing are mapped to edges attached to p and labelled with the process. A particular attention is necessary for **login** prefix because it needs to be connected to nodes p and s through an edge that will make **login** action to synchronize with an eventual corresponding **accept** action performed at s. The parallel processes are mapped to the union of the hypergraphs of their parallel components; finally, process invocations are translated with the unfolding of the definition where formal parameters are substituted with the actual ones. The translation for normal processes is analogous and, therefore, omitted.

6.3.2 Productions (no path reservation)

As anticipated above, hypergraphs allows path reservation. However, we prefer to introduce first productions which does not consider path reservation, but are more strictly related to QLAIM semantics and, later on, we show how a path can be reserved and traversed.

As done for the translation of Ambient presented in Section 5.1, we distinguish between *activity* and *coordination* productions.

Coordination productions for QLAIM are more sophisticated than coordination production for Ambient. Essentially, it is necessary to coordinate sites, gateways, processes and router edges in order to detect the best path connecting two nodes. Moreover, QLAIM also specifies a sophisticated mechanism for data exchanging. Indeed, after the discover of a path toward the tuple space that must be accessed, it is necessary to "find" the matching tuple. For this reasons, we separate the presentation of activity and coordination productions in different sections. Section 6.3.3 describes the activity productions necessary for executing QLAIM actions; Section 6.3.4, Section 6.3.5 and Section 6.3.6 respectively report coordination productions for QLAIM specific actions, for router edges and for gateway edges.

6.3.3 QLAIM activity productions

Activity productions for QLAIM regard the actions for accessing tuple spaces or for spawning remote processes.

All actions that refer a (possibly) remote site require an initial common phase: The search of a path of gateways to the destination node. Let a be one of the action $\mathbf{in}(T)$, $\mathbf{read}(T)$ or $\mathbf{eval}(Q)$:

$$p \vdash L_{a@s.\mathbb{P}}(p) \xrightarrow{\{(p,\ \overline{\mathbf{a}_a\ s},\ \langle\rangle)\}} p \vdash \underline{L}_{a@s.\mathbb{P}}(p), \tag{6.2}$$

where $\mathbf{a}_a = \begin{cases} \mathbf{in}, & \text{if } a = \mathbf{in}(T) \\ \mathbf{rd}, & \text{if } a = \mathbf{read}(T) \\ \mathbf{ev}, & \text{if } a = \mathbf{eval}(Q) \end{cases}$

Production (6.2) states that an edge corresponding to (possibly) remote actions emits on its interface node p a signal which corresponds to the action a and moves in a state ($\underline{L}_{a@s.\mathbb{P}}$) where the cost of a path to the remote node s is waited. When an effective cost $\kappa \neq \infty$ is obtained, the action is effectively "executed":

$$p \vdash \underline{L}_{a@s.\mathbb{P}}(p) \xrightarrow{\{(p,\ s\ \kappa,\ \langle z\rangle)\}} p, z \vdash H \tag{6.3}$$

where H is a graph that depends on a. If $a = \mathbf{eval}(Q)$ then $H = [\![\ \mathbb{P}\ ]\!]_p \mid [\![\ Q\ ]\!]_z$, namely the graph that corresponds to the continuation of the **eval** action continues its execution at p, while the graph corresponding to Q is attached to the remote node z which corresponds to the "p-node" of the remote site s.

The cases $a = \mathbf{in}(T)$ or $a = \mathbf{read}(T)$ are more complex, indeed H is the edge $\widehat{L}_{a@s.\mathbb{P}}(p,z)$ and new productions must be given to deal with tuple exchanging. The edge $\widehat{L}_{\mathbf{in}(T)@s.\mathbb{P}}(p,z)$ synchronizes with an output process (running at z). Once a tuple t that matches T is collected, it is "forwarded" to the process on p waiting for it

$$p, z \vdash \widehat{L}_{\mathbf{in}(T)@s.\mathbb{P}}(p,z) \xrightarrow{\{(z,\ \overline{in\ l_T},\ \langle\eta_t\rangle)\}} p, z, \eta_t \vdash [\![\ \mathbb{P}[{}^{t}/_{T}]\ ]\!]_p.$$

Action l_T depends on template T and is defined as follows:

T	v	s	$!x$	$!u$	F, T'
l_T	v	s	$v \in Val$	$s \in \mathcal{S}$	$l_F, l_{T'}$

This corresponds to QLAIM tuple matching thanks to the definition of l_T (in the cases $!x$ and $!u$). For $\mathbf{read}(T)$ prefix, we have similar productions: It is sufficient to replace **in** with **read**, $in\ l_T$ with $rd\ l_T$ and $\overline{\mathbf{in}}\ s$ with $\overline{\mathbf{rd}}\ s$ in the above productions.

The edge for an output process waits on its node for a synchronization action of an input or read action. It is necessary to consider site names that appear inside output tuples because they correspond to nodes in the graph that could be possibly extruded. Given a tuple t, we let η_t be the set of site names occurring as fields of t:

$$p \vdash L_{\mathbf{out}(t)}(p) \xrightarrow{\{(p,\ rd\ t,\ \langle\eta_t\rangle)\}} p \vdash L_{\mathbf{out}(t)}(p), \tag{6.4}$$

$$p \vdash L_{\mathbf{out}(t)}(p) \xrightarrow{\{(p,\ in\ t,\ \langle\eta_t\rangle)\}} p \vdash nil. \tag{6.5}$$

Production (6.4) synchronizes with read actions, indeed the continuation of the production still contain the output process. Production (6.5) deals with input actions. Note that the continuation does not anymore contain the output process.

In the production (6.3) relative to edge $\underline{L}_{a@s.\mathbb{P}}$, if no path of gateways exists between (the site relative to) p and s then the execution gets stuck because no other transition can be derived.

An edge corresponding to $\mathbf{newloc}(u, \mathbb{P}').\mathbb{P}$ performs a silent action and becomes a graph obtained by putting side by side the graph of a new site with $\mathbb{P}'$ as coordinator and the graph of the continuation $\mathbb{P}$.

$$p \vdash L_{\mathbf{newloc}(u,\mathbb{P}').\mathbb{P}}(p) \xrightarrow{\{(p,\tau,\langle\rangle)\}} p \vdash (\nu\ s).([\![\ \mathbb{P}[s/u]\ ]\!]_p \mid [\![\ \mathbb{P}'\ ]\!]_s),$$

where $s \notin \{p\} \cup \mathrm{fn}(\mathbb{P}, \mathbb{P}')$[1].

A different "scheme" is adopted for productions of coordinator process actions that are used for managing the network topology and, as stated before, are all local actions. In general, this productions synchronize with their site edge and demand to it the effective topology change. Once the request has been satisfied, an "acknowlegde" message makes the continuation process of the action to be activated.

[1]Note that, initially, no gateway connection is present between s and p.

Productions for **login**/**accept** actions set a gateway edge. The following productions state that the **login** edge first asks for the creation of a new gateway to s (label ng) with cost κ and then it waits for the gateway-connected signal (label gc) from its site edge

$$p, s \vdash \lambda_{\mathbf{login}(s,\kappa).\mathbb{P}}(p, s) \xrightarrow{\{(p,\ \overline{ng\ \kappa},\ \langle s\rangle)\}} p, s, x \vdash \lambda'_{\mathbf{login}(s,\kappa).\mathbb{P}}(x, p, s) \tag{6.6}$$

$$p, s, x \vdash \lambda'_{\mathbf{login}(s,\kappa).\mathbb{P}}(x, p, s) \xrightarrow{\{(p,\ gc,\ \langle\rangle)\}} x, p, s \vdash [\![\ \mathbb{P}\ ]\!]_p. \tag{6.7}$$

Notice that $\lambda_{\mathbf{login}(s,\kappa).\mathbb{P}}$ synchronizes on node p only. In other words, it interact only with its site edge.

Production for **accept** acts similarly to the production for **login**: The site node is asked to accept login connections coming from s (provided that the cost is κ):

$$p \vdash L_{\mathbf{accept}(s,\kappa).\mathbb{P}}(p) \xrightarrow{\{(p,\ \overline{s\ acc\ \kappa},\ \langle\rangle)\}} p \vdash L'_{\mathbf{accept}(s,\kappa).\mathbb{P}}(p)$$

$$p \vdash L'_{\mathbf{accept}(s,\kappa).\mathbb{P}}(p) \xrightarrow{(p,\ gc\ \langle\rangle)} p \vdash [\![\ \mathbb{P}\ ]\!]_p.$$

Then, as for **login**, once the gateway has been connected, it receives from its site edge the continuation signal (label gc).

Let a be either $\mathbf{logout}(s, \kappa)$ or $\mathbf{disc}(()s, \kappa)$; the productions below signal to their site edges the request for removing the gateway with cost κ that connects the local site to s and wait for the continuation signal gr:

$$p \vdash L_{a.\mathbb{P}}(p) \xrightarrow{\{(p,\ \overline{s\ det\ \kappa},\ \langle\rangle)\}} p \vdash L'_{a.\mathbb{P}}(p)$$

$$p \vdash L'_{a.\mathbb{P}}(p). \xrightarrow{\{(p,\ gr,\ \langle\rangle)\}} p \vdash [\![\ \mathbb{P}\ ]\!]_p.$$

6.3.4 Coordinating QLAIM actions

This section details the productions necessary for making site edges to coordinate requests of actions from their local processes. Hereafter we let **a** be an element of the set $\{\mathbf{in}, \mathbf{rd}, \mathbf{ev}\}$.

We start with the simplest case: If a process wants to access the local tuple space, or spawn a process on the local site, then there is no need for starting a routing search. In this case, the site edge can immediately reply to the process with a path that has the minimal cost. The following two productions formalize what informally stated:

$$p, r, s \vdash \mathfrak{S}_s(p, r, s) \xrightarrow{\{(p,\ \mathbf{a}\ s,\ \langle\rangle)\}} p, r, s \vdash \underline{\mathfrak{S}_s}(p, r, s) \tag{6.8}$$

$$p, r, s \vdash \underline{\mathfrak{S}_s}(p, r, s) \xrightarrow{\{(p,\ \overline{s\ \top},\ \langle p\rangle)\}} p, r, s \vdash \mathfrak{S}_s(p, r, s). \tag{6.9}$$

When (production (6.8)) the site edge recognizes that a local process wants to locally perform an action it moves in a state where (production (6.9)) it replies with the trivial cost-less path.

Observation 6.3.1 *This mechanism presupposes that processes can always access their local tuple spaces and spawn processes on their local execution site. Of course, this is arbitrary and other choices can be adopted. For instance, we can give productions that allows processes to access the local site only if some access control policy is satisfied.*

When a process must perform a remote action, the site edge intermediates between the process and router edges. Let us consider the following productions:

$$p, r, s \vdash \mathfrak{S}_s(p, r, s) \xrightarrow{\{(p,\ \mathbf{a}\ s',\ \langle\rangle), (r,\ \overline{s\ \mathbf{a}\ s'},\ \langle\rangle)\}} p, r, s \vdash \underline{\mathfrak{S}}_{s\mathbf{a}s'}(p, r, s)$$

$$p, r, s \vdash \underline{\mathfrak{S}}_{s\mathbf{a}s'}(p, r, s) \xrightarrow{\{(r, s'\ \kappa, \langle z\rangle), (p, \overline{s'\ \kappa}, \langle z\rangle)\}} p, r, s, z \vdash \mathfrak{S}_s(p, r, s);$$

the first production states that the request for a remote site $s' \neq s$ is forwarded to the router edge. Then the site edge waits for the answer of the router edge (on node r) and forward it to the waiting process. However it must be considered the possibility of cycles in the topology of the net. Therefore the intermediate state of the site edge $\underline{\mathfrak{S}}_{s\mathbf{a}s'}$ replies with an infinite cost to all requests for a path to s' for action $\mathbf{a}$ executed at s which are detected at node s. The following productions act as described:

$$p, r, s \vdash \underline{\mathfrak{S}}_{s\mathbf{a}s'}(p, r, s) \xrightarrow{\{(s,\ s\ \mathbf{a}\ s',\ \langle\rangle)\}} p, r, s \vdash S_{s\mathbf{a}s'}(p, r, s)$$

$$p, r, s \vdash S_{s\mathbf{a}s'}(p, r, s) \xrightarrow{\{(s, \overline{s'\ \infty}, \langle\rangle)\}} p, r, s \vdash \underline{\mathfrak{S}}_{s\mathbf{a}s'}(p, r, s).$$

Some care is necessary for handling the case when no finite-cost path is found. The problem is that, the site edge must not be blocked but, as prescribed by the QLAIM semantics, the process that issued the request gets stuck:

$$p, r, s \vdash \underline{\mathfrak{S}}_{s\mathbf{a}s'}(p, r, s) \xrightarrow{\{(r,\ s'\ \infty,\ \langle\rangle)\}} p, r, s \vdash \mathfrak{S}_s(p, r, s).$$

In other words, detection of infinite-cost path to a node makes the site edge to return in its initial state, while the process asking for the gateway path keeps waiting.

Finally, the last coordination production relative to site edges are those productions regarding action of coordination processes. Let us first consider the productions that synchronize with **login** edges. The establishment of a gateway takes place in two steps. The first step is devoted to synchronize the $ng\ \kappa$ request of the **login** edge and the second synchronize the remote site edge, the (local) router edge and the waiting **login** edge (see productions (6.6) and (6.7), page 111):

$$p, r, s \vdash \mathfrak{S}_s(p, r, s) \xrightarrow{\{(p,\ ng\ \kappa,\ \langle x\rangle))\}} p, r, s, x \vdash \widehat{\mathfrak{S}_s}(p, r, s, y)$$

$$p, r, s \vdash \widehat{\mathfrak{S}_s}(p, r, s, y) \xrightarrow{\{(y,\ \overline{s\ lin\ \kappa}, \langle x\rangle), (r,\ \overline{ng},\ \langle x\rangle), (p,\ \overline{gc},\ \langle\rangle)\}} p, r, s, x \vdash \mathfrak{S}_s(p, r, s)$$

Notice that, in the second production, the site edge synchronizes contemporary with all the edges of its interfaces.

Symmetrically, when a site receives both a message from one of its processes for accepting a gateway from a remote site s' having cost κ and a message on s from s' asking for creating a gateway, then it simply has to signal to the **accept** edge that the gateway has been created:

$$p, r, s \vdash \mathfrak{S}_s(p, r, s) \xrightarrow{\{(p,\ s'\ acc\ \kappa,\ \langle\rangle), (s,\ s'\ lin\ \kappa, \langle\rangle)\}} p, r, s \vdash \mathfrak{S}'_s(p, r, s)$$

$$p, r, s \vdash \mathfrak{S}'_s(p, r, s) \xrightarrow{\{(p,\ \overline{ng},\ \langle\rangle)\}} p, r, s \vdash \mathfrak{S}_s(p, r, s)$$

If a site edge receives a logout signal from one of its local processes, it forwards the message to the router edge in order to make the corresponding gateway to be detached.

$$p, r, s \vdash \mathfrak{S}_s(p, r, s) \xrightarrow{\{(p,\ s'\ det\ \kappa,\ \langle\rangle), (r,\ \overline{s'\ det\ \kappa},\ \langle\rangle)\}} p, r, s \vdash \mathfrak{S}_s(p, r, s)$$

Once the router edge has disconnected the gateway, a "gateway-removed" (gr) signal will be sent (on node r) to the site edge that would provide to forward it to the waiting **disc**( )edge.

$$p, r, s \vdash \mathfrak{S}_s(p, r, s) \xrightarrow{\{(r, gr, \langle\rangle), (p,\ \overline{gr},\ \langle\rangle)\}} p, r, s \vdash \mathfrak{S}_s(p, r, s)$$

6.3.5 Routing productions

A *Router edge* Δ is connected to each node representing a QLAIM site (through edge $\mathfrak{S}$). A gateway $\langle s', \kappa\rangle$, from s to s', is connected to the routing edge of s and reaches s'. A router edge $\Delta_n(\vec{x}, r)$ has an outgoing tentacle entering a node r and n tentacles where the gateways departing from s are connected to. Intuitively, router edges have the rôle of determining the optimal path of remote operations with respect to the QoS attributes specified by QLAIM networking constructs. Searching the optimal path is a distributed operation and each router edge receives from its neighbors the current information about the optimal paths. A router edge $\Delta_n(\vec{x}, r)$ receives the result of the search of the optimal path with respect to the QoS attributes at x_1,..., x_n. Then, it selects one tentacle among those that return the "minimal" cost and propagates the result to r.

Basically, the productions of the router edge provide a declarative specification of the standard distance vector algorithm which computes the optimal route inside a network. Indeed, the productions specify the constraints and the selection of the productions to compute the optimal path requires the solution of a distributed constraint solving algorithm. We refer to [136] for a detailed description of the relationships between graph rewriting systems and constraint solving algorithms.

We now present the productions which detail the behaviour of router edges. The first production forwards to gateway edges the requests received from the site:

$$r, \vec{x} \vdash \Delta_n(\vec{x}, r) \xrightarrow{\{(r,\ s\ \mathbf{a}\ s',\ \langle\rangle),\ (x_1,\ \overline{s\ \mathbf{a}\ s'}, \langle\rangle), ..., (x_n,\ \overline{s\ \mathbf{a}\ s'},\ \langle\rangle)\}} r, \vec{x} \vdash \underline{\Delta}_n^{s\mathbf{a}s'}(\vec{x}, r).$$

This production states that $\Delta_n(\vec{x}, r)$ receives on r a signal for a connection from s to s' with respect to an action $\mathbf{a}$. The router edge forwards the signal along its tentacles $x_1, ..., x_n$ and evolves in the state $\underline{\Delta}_n^{s\mathbf{a}s'}$ that waits for results of the activated searches:

$$r, \vec{x} \vdash \underline{\Delta}_n^{s\mathbf{a}s'}(\vec{x}, r) \xrightarrow{\left\{\begin{array}{c}(x_1,\ s'\ \kappa_1,\ \langle u_1\rangle),\\ ...,\\ (x_n,\ s'\ \kappa_n,\ \langle u_n\rangle),\\ (r,\ \overline{s'\ \kappa_h},\ \langle u_h\rangle)\end{array}\right\}} r, \vec{x}, \vec{u} \vdash \Delta_n(\vec{x}, r), \tag{6.10}$$

where $\kappa_h = min\{\kappa_1, ..., \kappa_n\}$ and $\vec{u} = u_1, ..., u_n$. $\underline{\Delta}_n^{s\mathbf{a}s'}$ receives the results along its incoming tentacles, and computes the optimal cost (κ_h). Then, it forwards node u_h to r and restarts from the initial state $\Delta_n(\vec{x}, r)$.

The router edge can receive a "new-gateway" request (ng) from its site edge. In this case it creates a private node x that is fused with the node offered by the complementary production,

$$r, \vec{x} \vdash \Delta_n(\vec{x}, r) \xrightarrow{\{(r,\ ng,\ \langle y\rangle)\}} r, \vec{x}, y \vdash \Delta_{n+1}(\vec{x}, y, r).$$

Node y is used as the node where a tentacle of Δ_n and the gateway are connected.

If a site edge receives a detach signal from one of its local processes, it forwards the message to the router edge that, further forwards the message to its gateways:

$$\vec{x}, r \vdash \Delta_n(\vec{x}, r) \xrightarrow{\{(r,\ s\ det\ \kappa,\ \langle\rangle), (x_1,\ \overline{s\ det\ \kappa},\ \langle\rangle), ..., (x_n,\ \overline{s\ det\ \kappa},\ \langle\rangle)\}} \vec{x}, r \vdash \Delta'_n(\vec{x}, r)$$

The router edge must wait for the gateway to s with cost κ to reply its disconnection from s so that the completion of the detaching operation can be back forwarded to the process that triggered it. This behaviour is specified by the following production:

$$\vec{x}, r \vdash \Delta'_n(\vec{x}, r) \xrightarrow{\{(x_1,\ nodet,\ \langle\rangle), ..., (x_i,\ gr\ ,\langle\rangle), ..., (x_n,\ nodet,\ \langle\rangle), (r,\ \overline{gr},\ \langle\rangle)\}} \vec{z}, r \vdash \Delta_{n-1}(\vec{z}, r)$$

(where $\vec{z} = x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$).

6.3.6 Gateway productions

Whenever a gateway $G_s^\kappa(x, s)$ receives a message for searching a path to a node s' ($s' \neq s$) for action $\mathbf{a}$, then it forwards the signal, provided that $\kappa \models \mathbf{a}$:

$$x, s \vdash G_s^\kappa(x, s) \xrightarrow{\{(x,\ q\mathbf{a}s',\ \langle\rangle),\ (s,\ \overline{q\mathbf{a}s'},\ \langle\rangle)\}} x, s \vdash \widehat{G}_s^{s',\kappa}(x, s).$$

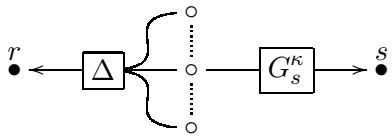
Edge $\widehat{G}_s^{s',\kappa}(x, s)$ waits on s for the cost κ' of the path from s to s' and sends back to the router edge the new value of the optimal path.

$$x, s \vdash \widehat{G}_s^{s',\kappa}(x, s) \xrightarrow{\{(s,\ s'\ \kappa',\ \langle u\rangle), (x, \overline{s'\ \kappa' \oplus \kappa},\ \langle u\rangle)\}} x, s, u \vdash G_s^\kappa(x, s) \tag{6.11}$$

Otherwise, if $\kappa \not\models \mathbf{a}$, then the infinite cost is backward propagated.

$$x, s \vdash G_s^{\kappa}(x,s) \xrightarrow{\{(x,\ q\mathbf{a}s',\ \langle\rangle)\}} x, s \vdash \widehat{G}_s^{s',\infty}(x,s)$$
$$x, s \vdash \widehat{G}_s^{s',\infty}(x,s) \xrightarrow{\{(x,\ \overline{s'\,\infty},\ \langle\rangle)\}} x, s \vdash G_s^{\kappa}(x,s).$$

Given a hypergraph $\Gamma \vdash G$, we say that nodes r and s of G are *gateway-adjacent* if the graph below is a subgraph of $\Gamma \vdash G$.



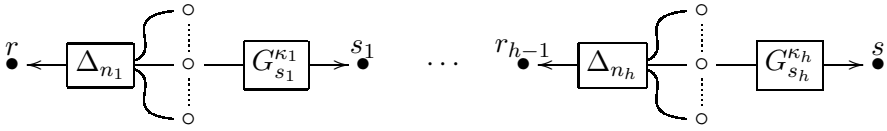
A *gateway path in* G is a sequence of gateway-adjacent nodes; we say that (free) nodes of a gateway path are *gateway-connected*. The cost of a gateway path is the sum of the costs associated to each gateway edge appearing in the path. We can now state an important result on selecting the minimal cost path between two gateway-connected nodes.

Theorem 6.3.1 *Let $\Gamma \vdash G$ be a hypergraph and $r, s \in \Gamma$. Consider the transition*

$$\Gamma \vdash G \xrightarrow{\Lambda \cup \{(r,\, s\ \kappa,\, \langle u\rangle)\}} \Gamma' \vdash G', \tag{6.12}$$

then following statements hold:

1. *if transition (6.12) can be derived then r and s are gateway-connected by a path costing κ;*
2. *if*



is a gateway-path between r and s in G, then there is a transition like (6.12) such that $\kappa \leq \Sigma_{i=1}^{h} \kappa_i$.

Proof. Statement *1* of the theorem is evident if we consider the interplay between the productions of routing and gateway edges. Indeed, routing egdes have productions that signal cost messages $s\ \kappa$ only if the receive them on one of their tentacles which are connected to gateway edges. Moreover, according to production (6.11), each gateway edge in the path backwardly forwards the sum of its cost and the minimal cost received along its outgoing tentacle. Having observed this, if we proceeds by induction on the structure of the proof of transition (6.12) we obtain the thesis.

The second part of the theorem is also evident. Reasoning by absurd, if there is a gateway path having cost strictly less than κ, then production (6.10) would have filtered this cost when the synchronizations on its incoming tentacles would have took place.

Hence these synchronizations would have discharged the path of cost κ, which is contrary to our hypothesis. □

Theorem 6.3.1 means that the path search triggered by remote actions detects a gateway-path if it exists in the graph (first part of the theorem), moreover the search always selects the minimal cost path connecting two gateway-nodes (second part of the theorem).

Finally, we must consider the productions for disconnecting gateways. When gateway edges receive the logout signal from their router edge, they simply disappears:
small

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{\{(x,\ s\ det\ \kappa,\ \langle\rangle)\}} x, s \vdash D_s^\kappa(x,s)$$

$$x, s \vdash D_s^\kappa(x,s) \xrightarrow{\{(x,\ gr,\ \langle\rangle)\}} s \vdash nil$$

Notice that, the right-hand-side of the last production, the (private) node x disappear. On the other hand, if the gateway is not the gateway selected by the logout signal, the gateway edge remains attached to the router edge:

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{\{(x,\ s'\ det\ \kappa',\ \langle\rangle)\}} x, s \vdash N_s^\kappa(x,s)$$

$$x, s \vdash N_s^\kappa(x,s) \xrightarrow{\{(x,\ \overline{nodet}\ \langle\rangle)\}} x, s \vdash G_s^\kappa(x,s).$$

where $s \neq s'$ or $\kappa \neq \kappa'$.

6.4 Productions for path reservation

This section aims at modifying the productions presented so far in order to permit path reservation and "routing" along reserved path of information necessary for QLAIM computations that require remote accesses. We will show how path reservation is essentially obtained by enriching the behaviour of routing and gateway edges with new productions and with slight variations of productions for QLAIM actions introduced in Section 6.3.3.

Let us again consider production (6.10) that we report here:

$$r, \vec{x} \vdash \underline{\Delta}_n^{s\mathbf{a}s'}(\vec{x}, r) \xrightarrow{\{(x_1,\ s'\ \kappa_1,\ \langle u_1\rangle), ..., (x_n,\ s'\ \kappa_n,\ \langle u_n\rangle), (r,\ \overline{s'\ \kappa_h},\ \langle u_h\rangle)\}} r, \vec{x}, \vec{u} \vdash \Delta_n(\vec{x}, r).$$

where $\kappa_h = min\{\kappa_1, ..., \kappa_n\}$ and $\vec{u} = u_1, ..., u_n$. As stated above, this production forwards (through the site edge) the minimal cost computed to the process edge that want to perform a remote action. In order to reserve paths it is necessary communicate to the gateway edges whether they are reserved or not. Therefore, we replace the previous production with

$$r, \vec{x} \vdash \underline{\Delta}_n^{s\mathbf{a}s'}(\vec{x}, r) \xrightarrow{\{(x_1,\ s'\ \kappa_1,\ \langle u_1\rangle), ..., (x_n,\ s'\ \kappa_n,\ \langle u_n\rangle), (r,\ \overline{s'\ \kappa_h},\ \langle u_h\rangle)\}} r, \vec{x}, \vec{u} \vdash \widehat{\Delta}_n^h(\vec{x}, r).$$

The edge $\widehat{\Delta}_n^h$ represents an intermediate state of routing edges that must inform its gateway edges whether they are reserved or not:

$$r, \vec{x}, \vec{u} \vdash \widehat{\Delta}_n^h(\vec{x}, r) \xrightarrow{\Lambda} r, \vec{x}, \vec{u} \vdash \Delta_n(\vec{x}, r),$$

where $\Lambda = \{(x_i,\ \overline{nores},\ \langle\rangle) : i = 1, ..., n\ \wedge\ i \neq h\} \cup (x_h,\ \overline{res},\ \langle\rangle)$; this production makes $\widehat{\Delta}_n^h$ to communicate to the h-th gateway that it is reserved and to the remaining edges that they have not been selected. Of course gateways must interact with router edges in order to accomplish previous productions. In particular, production (6.11) (see page 114) must be changed with

$$x, s \vdash \widehat{G}_s^{s',\kappa}(x, s) \xrightarrow{\{(s,\ s'\ \kappa',\ \langle u\rangle), (x, \overline{s'\ \kappa' \oplus \kappa},\ \langle u\rangle)\}} x, s, u \vdash Pr_s^\kappa(x, s, u).$$

The difference with respect to production (6.11) is that, once the gateway has backward propagated the cost, it moves to a state Pr_s^κ where the *nores*/*res* signal is waited. Edge $Pr_s^\kappa(x, s, u)$ has an incoming tentacle from x, two outgoing tentacles to s and u (where u represents the next hop node). Notice that such node has been communicated during cost propagation.

If a *noact* signal is received then Pr_s^κ becomes the gateway to s as stated in the following production:

$$x, s, u \vdash Pr_s^\kappa(x, s, u) \xrightarrow{\{(x,\ noact,\ \langle\rangle)\}} x, s, u \vdash G_s^\kappa(x, s).$$

Otherwise, a packet will be attached to s and Pr_s^κ will take care of its destination. If the destination is s, the packet will terminate its travel:

$$x, s, u \vdash Pr_s^\kappa(x, s, u) \xrightarrow{\{(x,\ dest\ s,\ \langle\rangle)\}} x, s, u \vdash \underline{Pr}_s^\kappa(x, s, u)$$

$$x, s, u \vdash \underline{Pr}_s^\kappa(x, s, u) \xrightarrow{\{(x,\ \overline{stop},\ \langle u\rangle)\}} x, s, u \vdash G_s^\kappa(x, s).$$

Once Pr_s^κ receives a signal from a packet that wants to reach s it replies with a *stop* message where the last hop node is communicated. The intention is that u is the p-node of the site edge of s.

A *jump* signal is emitted, to let the packet reach node s' different from s:

$$x, s, u \vdash Pr_s^\kappa(x, s, u) \xrightarrow{\{(x,\ dest\ s',\ \langle\rangle)\}} x, s, u \vdash \widehat{Pr}_s^\kappa(x, s, u)$$

$$x, s, u \vdash \widehat{Pr_s^\kappa}(x, s, u) \xrightarrow{\{(x,\ jump,\ \langle u\rangle)\}} x, s, u \vdash G_s^\kappa(x, s).$$

The final changes regard QLAIM activity productions that has been introduced in Section 6.3.3. More precisely, let us consider production (6.3) reported below:

$$p \vdash \underline{L}_{a@s.\mathbb{P}}(p) \xrightarrow{\{(p,\ s\ \kappa,\ \langle z\rangle)\}} p, z \vdash H,$$

graph H must be substituted with a graph that models a packet containing the graph that must be routed at the remote site. As before, H depends on the action a; in the case of a being $\mathbf{eval}(Q)$, H is the graph $[\![\, \mathbb{P} \,]\!]_p \mid \nu\ u.([\![\, Q \,]\!]_u \mid Pk_s(u,z))$, whereas, if a is an input ($\mathbf{in}(T)$) or a read prefix ($\mathbf{read}(T)$), then H is $\nu\ u.(\widehat{L}_{a@s.\mathbb{P}}(p,u) \mid Pk_s(u,z))$.

The packet edge Pk_s has an incoming and an outgoing tentacle. The incoming tentacle insists on a private node u where the graph that must reach s is attached to; while the outgoing tentacle is initially connected to the first node of the route and successively it will correspond to the remaining nodes of the path, until the last node will be reached. On the final node the packet edge will be "disclosed" and its content, namely the graph connected to the incoming node, will be attached to the destination node. The packet edge $Pk_s(u,z)$ interacts with gateway edges; it communicates its destination and waits for a *stop* or a *jump* signal.

$$u, z \vdash Pk_s(u,z) \xrightarrow{\{(z,\ \overline{dest\ s},\ \langle\rangle)\}} u, z \vdash \underline{Pk_s}(u,z).$$

Next production below deals with the reception of a *jump* signal. In this case the packet will continue its execution on the next-hop node z':

$$u, z \vdash \underline{Pk_s}(u,z) \xrightarrow{\{(z,\ jump,\ \langle z'\rangle)\}} u, z, z' \vdash Pk_s(u,z').$$

If a *stop* signal is received (production (6.13)) Pk_s "dissolves" and fuses u on the received node p:

$$u, z \vdash \underline{Pk_s}(u,z) \xrightarrow[[p/u]]{\{(z,\ stop,\ \langle p\rangle)\}} p, z \vdash nil. \tag{6.13}$$

Notice that in the right-hand-side graph of production (6.13) the hypergraph attached to u is moved on p, the remote node where its execution continues.

Observation 6.4.1 *In order to keep the presentation as smooth as possible, in this section we have assumed that site edges takes care of not starting path searches when a request for the local site is issued (see productions (6.8) and (6.9) in Section 6.3.4). For the sake of precision, we must also deal with the case that packet Pk_s remains on node p of its site edge. There are two possibilities for overcoming this problem; the first possibility is to extend productions of site edges that, after synchronization of the production (6.9) also send a $(\_,\ stop,\ \langle p\rangle)$ signal to the packet edge. The second possibility is to enrich graphs of* Qlaim *nets by adding to each p-node of site edges a further edge that "continuously" emits the $(\_,\ stop,\ \langle p\rangle)$ signal for local packages. The second solution should be preferred to the first one for at least two reasons:*

- *It is not necessary to modify site edge productions, and*
- *it also simplifies productions of gateway edges because they should not worry about sending the stop signal.*

Productions presented in this section and Theorem 6.3.1 (in the previous section) ensure that, whenever a remote operation is performed, semantics of hypergraph always select the optimal path with respect to the QoS attributes specified by the QLAIM networking constructs.

This result depends on the outcome of a distributed constraint satisfaction problem, the *rule matching problem* [136]. For the result to hold, QoS attributes must form an ordered semi-ring [20], where the additive and multiplicative operations allow us to compare and compose QoS parameters.

Chapter 7

Hypergraphs and Software Design

Abstract

Traditional software engineering technologies emphasize an interaction model which is rather different from the interaction model of truly distributed applications. For instance, users of traditional distributed applications can invoke a service regardless of whether the service is local, remote or under the control of a different network authority. On the other hand, network *awareness* is crucial in WAN applications.

UML provides a widely accepted graphical notation to describe both structural and behavioral aspects of systems. This chapter describes a variation of graph transformation semantics which directly supports network awareness. Hence, what is missing in the UML specification can be actually found at the semantic level. We describe how hypergraphs can be used for specifying and refine software components constituting WAN applications.

Contents

7.1 Designing WAN Applications

Traditional software engineering technologies (e.g. client-server architecture) emphasize an interaction model which is rather different from the interaction model of truly distributed applications. For instance, users of traditional distributed applications can invoke a service regardless of whether the service is local, remote or under the control of a different network authority. Instead, in the context of Wide Area Network applications the *awareness* of network information is crucial for choosing the best services that match user's requirements. Indeed, network awareness can be exploited to provide as much information about the network facilities as possible to designers, aiming at specifying and implementing robust modules. On the other hand, in the next few years evolutionary *middlewares* based on SOAP-XML-UDDI-WSDL will probably become the standard in software industry. It is interesting to note, however, that some innovative applications (e.g. peer-to-peer) are developed largely "ad hoc", exploiting the traditional client-server interaction model.

The Unified Modeling Language (UML) [21, 141] has been widely accepted throughout the software industries and has become the *de facto* standard for specifying the development of software systems. In fact, UML provides a graphical notation to describe both structural and behavioral aspects of systems. In particular, class and state diagrams are the fundamental units which allow the designer to specify the behaviour of object-based systems. However, class and state diagrams provide the abstraction to understand method invocation independently from the location of the object. However, as pointed out in [105], method invocation in a truly distributed application is inherently different from method invocation in a traditional distributed application. This observation implies that eventually distributed issues *must* be taken into account. A specification technique which ignores such a difference will not support at the right level of abstraction software design pointing out the possible architectural choices in the system under development.

Previous work on the formalization of UML has produced a semantic framework based on *graph transformations* (see [67, 88, 111] and the references therein). The evolution of a UML specification may be understood as a graph transformation. This chapter describes a variation of graph transformation semantics which directly supports network awareness. Hence, what is missing in the UML specification can be actually found at the semantic level.

Hence, independently from the underlying technology, we argue that requirement engineering technologies must support the shift from the client-server interaction model to other interaction models which better accommodate the constraints posed by the new applications. The present chapter intends to address this issue. We describe how hypergraphs can be used for specifying and refine software components constituting WAN applications.

In other words, graphs and graph synchronization foster a declarative approach by identifying the points where satisfaction of certain properties has a strong impact on behaviours. The key issue of the approach is that components see the network

environment as a set of constraints. Then, the declarative specification of service requests to the network yields various kinds of constraints for the graphical calculus. Thus the actual behaviour is the result of a distributed constraint solving algorithm [136, 177].

We delineate a formal methodology that builds over graph synchronization to equip UML with semantical mechanisms to deal with the model ling of Wide Area Network applications.

7.2 Designing Software Using Hypergraph

In this section we show how synchronized edge rewriting can be exploited in the various phases of software development. In particular, we will consider UML [141] specifications and their graph transformation semantics as given in [111]. We first outline the main ideas of the methodology introduced in [111].

The drive-through example A drive-through can be visited by an ordered set of clients. Each client has a running number which indicates his/her turn. A client may submit an order to the drive-through that later will be served. The service order is established by the running number assigned at visit time.

Class, object and state diagrams are fundamental for UML specifications. Here we briefly illustrate how class and state diagrams can model the drive-trough example. For further details, the reader is referred to [21, 159, 141]

A UML *class diagram* describing the main relations among the component (i.e. the classes) of the system may be depicted as in Figure 7.1. The class diagram is Figure 7.1 represents the static structure of the system. Essentially, it provides information on the classes and relationships where the latter are divided into associations, generalizations, and dependencies. Special kinds of associations are compositions and aggregations. A class consists of a name, a set of attributes and a set of operations possibly with parameters. An *association end* is a language element of class diagrams which relates associations with classes and contains some information such as the rôle a class plays in the corresponding association or its multiplicity. A class diagram is a graph where the nodes are classes, and the edges are associations, generalizations, or dependencies. The drive-through class diagram models a client-server system containing a class DriveThrough and a class Client. A drive-through can be visited by an ordered set of clients (client queue). Every client of a drive-through has a running number which indicates the place the client has in the client queue of the DriveThrough. Furthermore, there is a class Order which represents possible orders a client can give to a drive-through.

Other features of systems are expressed in UML by means of *object diagrams* that may be thought of as diagrams describing the state of the system at a given moment. Figure 7.2 displays an object diagram of a possible evolution of the system described in Figure 7.1; a drive-through and three clients have been instantiated. Two of the

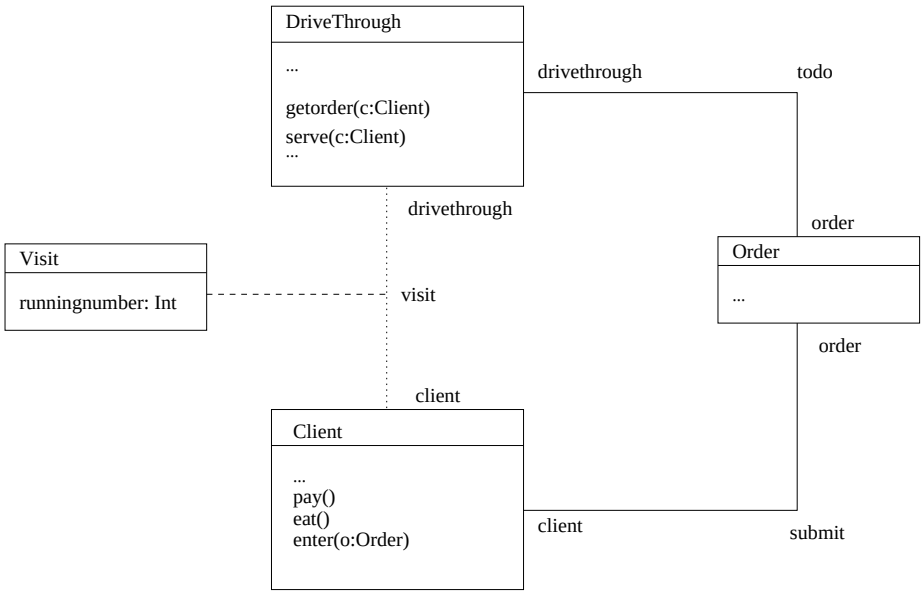


Figure 7.1: DriveThrough class diagram

clients visit the drive-through and one of them has issued an order. The operations

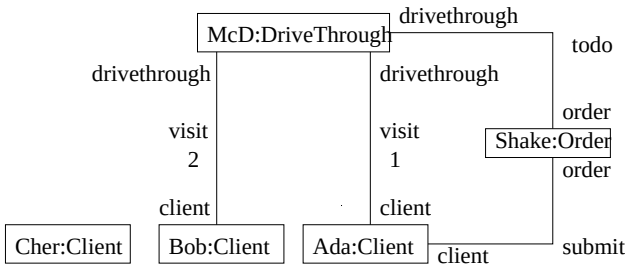


Figure 7.2: DriveThrough object diagram

listed in the class diagram may affect the relations among the objects in a given state of the system's evolution. This is captured by transformation rules related to class diagrams. These rules transforms object diagrams in object diagrams. In general, a set of graph transformations is associated to each specified operation. Figure 7.3 illustrates the rule of the serve operation for drive-through objects. The serve rule expresses that the link between the instance of an order and the instance of the drive-through that processes, it is removed when the serve action is executed.

Dynamic behaviour of the system's components is described in terms of *state diagrams*. State diagrams are associated to classes and describe the state changes of their objects. They are finite state automata whose transitions are labelled with an event, a guard and an action. Labels are written as $e[g]/o'.e'$, where e is the event that triggers the transition, g is a logic formula specified in OCL [141] and represents a pre-condition to the firing of the transition. Finally, $o'.e'$ is the invocation of the

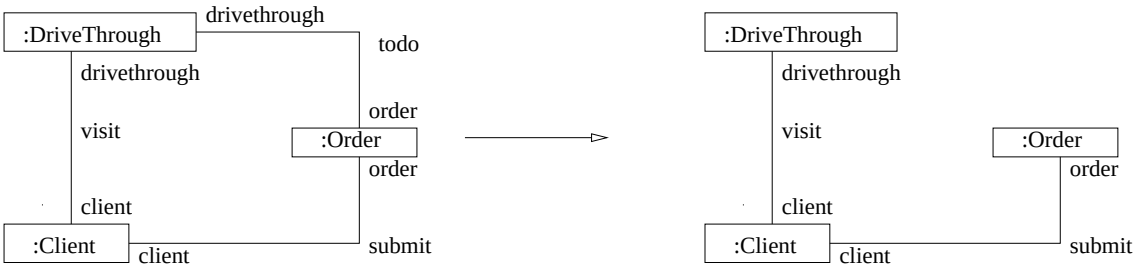


Figure 7.3: Serve operation

method e' of object o'.

Figure 7.4 describes the state diagrams of classes DriveThrough and Client. The

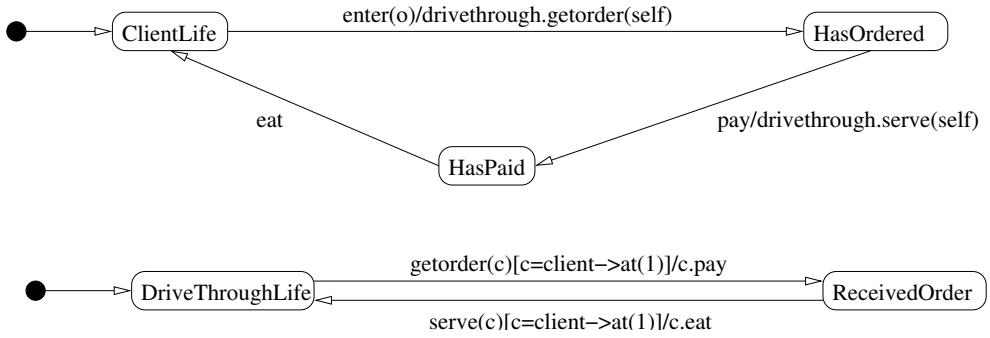


Figure 7.4: UML state diagram

Client diagram details the activity of a client as a cyclic sequence of entering an order/asking for the order to be executed, paying/waiting for being served and eating. When a drive-through must process an order, it checks that the order has been issued by the client on the top of the stack. In this case, the client is asked to pay for it and eventually the client is served and can start eating provided that payment has been performed.

Given a state diagram, it is possible to associate a graph transformation to each transition of the diagram. For this purpose, we assume that event stacks are associated to objects. Let us consider a transition $t = s \xrightarrow{e[g]/o'.e'} s'$ of a state diagram of class C. We may interpret t as the evolution of each object o in C whose first event in its event stack is e and the guard $[g]$ is evaluated to true; transition t also dispatches the event e' to the event stack of object o'. This interpretation may naturally be formalized with the graph transformation in Figure 7.5 while Figure 7.6 is an instance of the schema detailed above and describes the rule corresponding to the serve transition of the drive-through state diagram.

Roughly, the object transformations represent the global evolution of the system caused by the activity of its components, while transitions of state diagrams represent the local state changes. The graph transformation rules corresponding to those

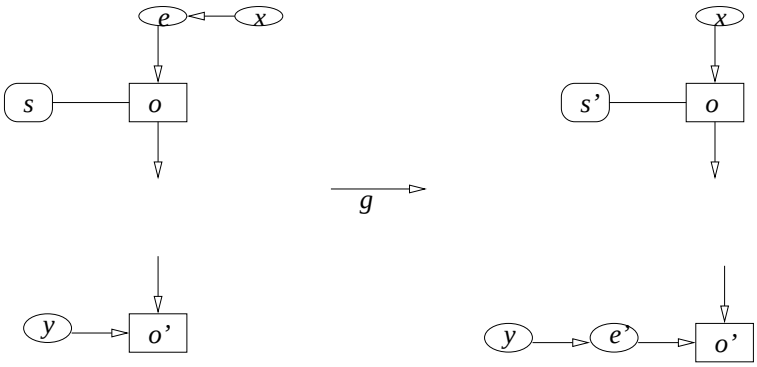


Figure 7.5: Graph Transformation of a Transition

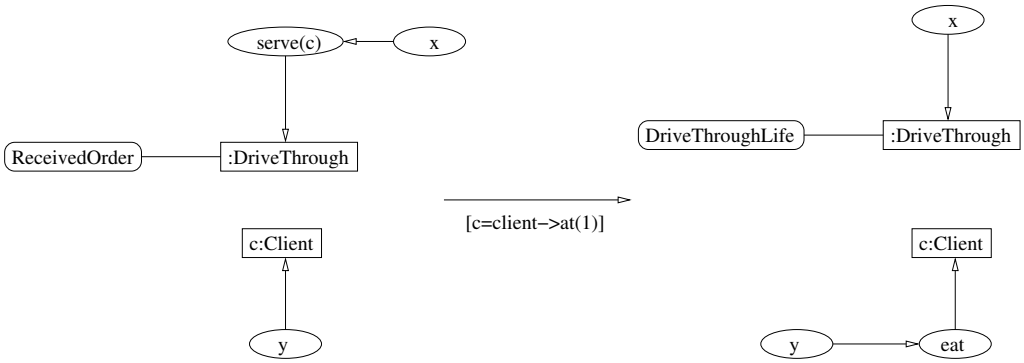


Figure 7.6: Transition rule for serve

different facets of system evolution must be mixed together in order to obtain the so called *integrated rules*. In the case of the serve rules, we have Figure 7.7.

Notice that the rule above do not specify some aspects that should be detailed in a complete specification. For instance, the integrated rule of Figure 7.7 does not describe how the eat event is pushed on the event stack of the client. In some sense, the interactions between the client and the drive-through remain to the abstract level of method invocation, without considering lower level aspects such as distribution or communications among objects.

7.3 Formal specification with edge replacement

In this section we describe how it is possible to associate in a uniform way productions of our calculus to the graph transformation rules given previously. We aim at showing the use of edge synchronization to formalize some issues that in the above specification have not been considered.

We consider three different forms of edges; events, controls and objects. They

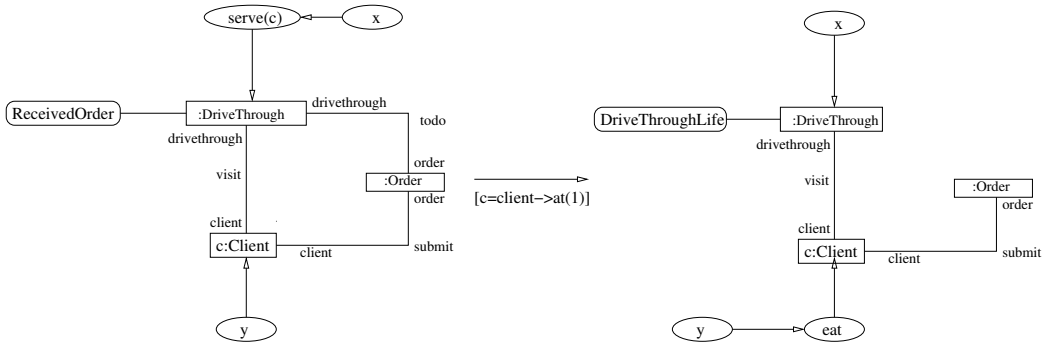
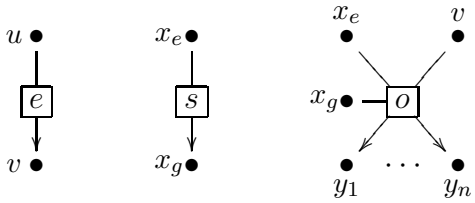


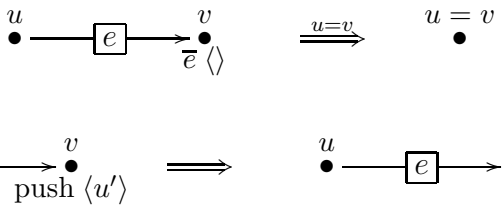
Figure 7.7: Integrated rule for serve

are graphically represented as



We assume that an edge label e does exist for each event e, and that a control edge exists for each state in a state diagram. Similarly, an object edge exists for each class in the UML specification. Edge e has two nodes such that a stack may be formed by merging node u of an edge labelled by e with a node v of another event edge. However, v nodes may also be fused with v of object edges. A control edge has two nodes. Node x_e is used to acquire the actual event from the object edge, while node x_g is used for checking guard satisfaction. These nodes are fused with the corresponding nodes of an object edge. An object edge has nodes for synchronizing with its control and event edges but also nodes $y_1, ..., y_n$ for connections with other objects according to the UML class diagram of the system.

Event edges must be popped when they synchronize with objects, and they must be pushed on the existing stack when they are created. Thus event edges have two productions; the first synchronizes with objects sending to them the event name. After the transition, the edge disappears and reconnects the rest of the stack with the v node of the corresponding object by fusing u and v. The second reacts to a "push" message:

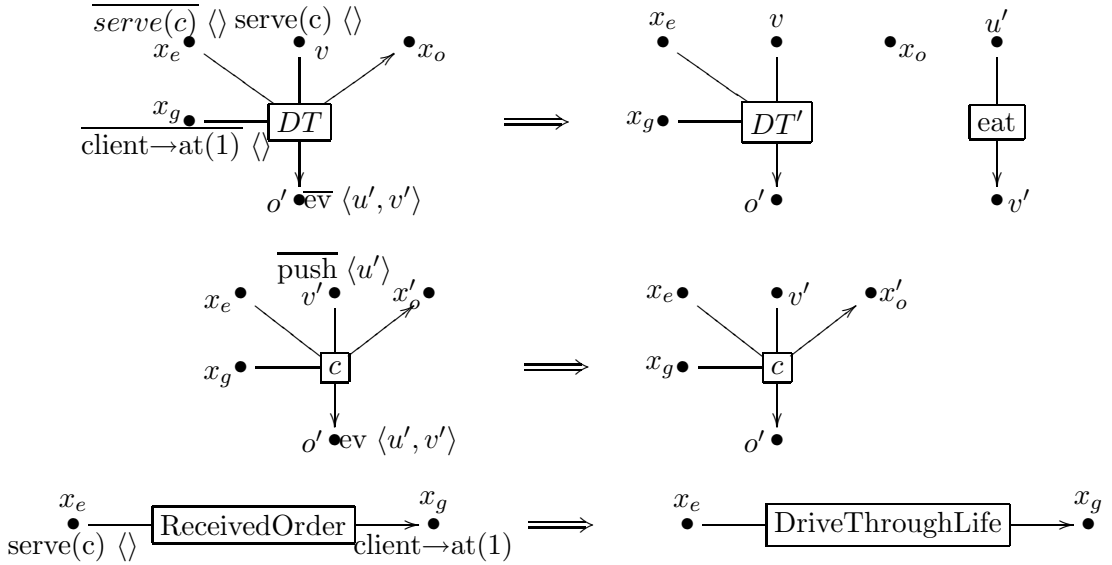


Note that the event that receives a push synchronization shifts back and fuses the v' node with the v node of the relative object edge. We remark that the previous

productions are obtained by considering the intended semantics of event stacks in the UML specification.

On the other hand, productions for control and object edges may be derived from the UML class, object and state diagrams in a uniform way.

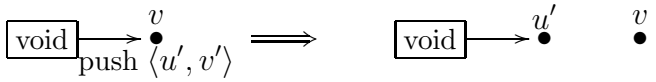
Let us consider the rules for serve in Figure 7.3 and 7.6. The following productions describe the evolution of each component of the system in terms of hypergraphs.



The first production states that an object edge DT that receives an event 'serve(c)' on the node corresponding to the event stack, evaluates the guard 'client→at(1)' and forwards the signal together with the evaluated guard to its control edge. It also sends the new event 'eat' on the node o' connected to the client; this is obtained by passing to the client object the nodes of the 'eat' event. As stated before, guards are expressed as OCL formulas; however, we do not model how they can be mapped into graphs and how they can be evaluated using edge replacement. The second production is the complementary rule of the previous production: when the client object receives the 'eat' event, it pushes the event and its stack. The last production states that the control edge 'ReceivedOrder' changes its label to 'DriveThroughLife' when the 'DriveThrough' object signals the 'serve(c)' event and the verification of the corresponding guard 'client→at(1)'.

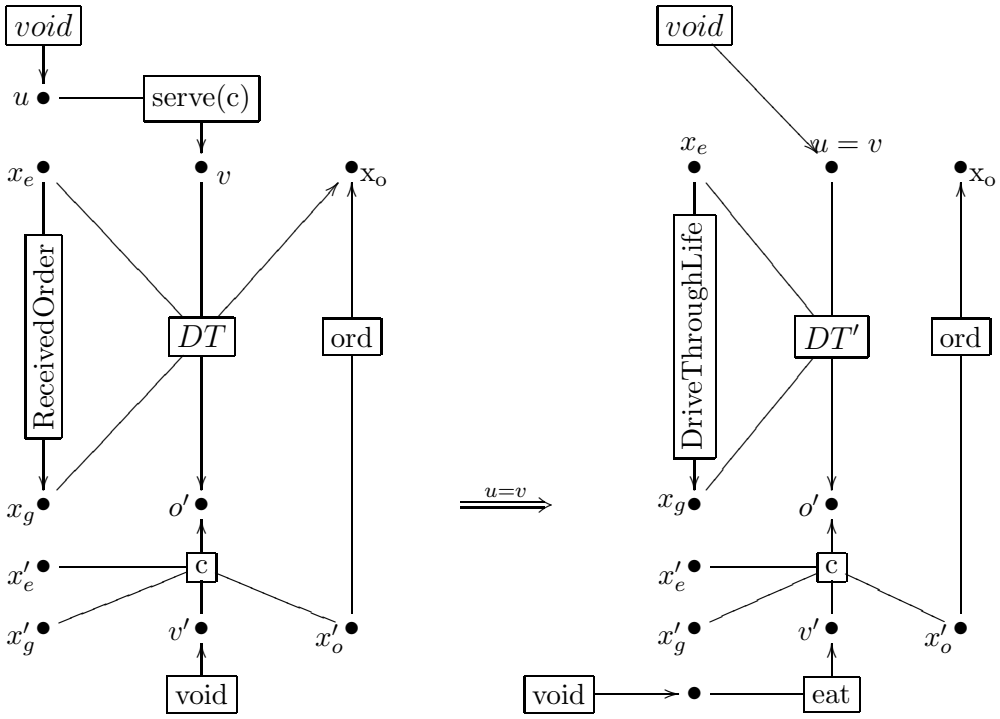
The productions introduced above guarantee that it is possible to obtain a transition which is equivalent to the integrated graph transformation in Figure 7.7.

In order to make the stack of events properly work, it is necessary to have an edge that manages the empty stack and allows an object edge to synchronize on push action only.



The edge behaves as an event edge that receives a push signal and, after the transition, it is connected to the node u' that is the last node of the stack of events.

The synchronization rules ensure that the following transition can be derived



Note also that the proof technique used to obtain it is as described for Ambient calculus in Chapter 5.

Part II

Security

Abstract

One of the most difficult issue to face when application must be executed in WAN's is *security*. In general, security aspects must be considered at any level, e.g. at the very low level of communication infrastructure as well as at the application level. At each level security assumes different meanings. For instance, at the application level it is typically intended as access control for protecting resources from undesired misuses.

At the level of communication infrastructure it is necessary to bear in mind that remote interactions take place along "public channels", namely, it is assumed, according to the Dolev-Yao model, that everyone can access the public communication media and interfere in any communication.

This part of the thesis introduces a framework for studying and analyzing cryptographic protocols. The framework is composed of a π-calculus-like process calculus and a logic. Our process calculus is well suited for specifying participants of protocol sessions. The logic is very simple and permits to specify security properties in terms of relationships among variables of the process terms. We show how multi-sessions can be simply managed in the framework both for specifying computations where many instances of the participants can be activated and for verifying security properties expressed in our logic.

Chapter 8

Security: an Overview

Abstract

This chapter collects and resumes some basic notion of cryptography, protocol specification and related security properties.

The chapter does not contain any original result but simply an overview of security issues in order to fix some terminology and notation. For both of them, we have tried to adopt standard and widely accepted conventions, therefore, the reader acquainted with such topics can safely skip the chapter.

Contents

8.1 Basic Notions of Cryptography

This section is devoted to review some elementary notions on cryptography. For a complete presentation we refer to [167, 123].

It is important to state which are the assumption adopted because correctness of a system with respect to security properties requires robustness at various abstraction level of the system. For instance, we will consider the underlying cryptographic system as "perfect", in the sense that the attacker cannot rely on flaws of encryption/decryption algorithms or on the representation of keys.

An *intelligible* message m, is referred to as *plaintext* (or *datagram*). By 'intelligible' it is usually intended that the representation of the information denoted by m is public domain knowledge. Viceversa, an *unintelligible* form of m is said *ciphertext* (or *cryptogram*).

The process of assigning a ciphertext to a plaintext is called *encryption*; usually, encryption is parameterized with respect to an *encryption key*. Given a ciphertext, the operation that reconstructs the plaintext form is called *decryption*; as for encryption, decryption usually has a *decryption key* as parameter. It is common notation to write $\{m\}_k$ for the ciphertext obtained by encrypting m under k, while k^{-1} denotes the corresponding decryption key.

Cryptography has been a requirement from ancient times and many cryptographic algorithm have been proposed during centuries. Most of them relied on secrecy of encryption and decryption keys (that usually were equal) and secrecy of encryption and decryption functions because the aim of those algorithms was to exchange informations between two intended partner which had the possibility of agreeing on secret keys and encryption/decryption mechanisms.

Modern day cryptography has to face the problem of making many users to communicate information each other in a reliable way over an untrusted medium and, therefore, it relies only on key secrecy. Indeed, modern cryptography follows the *Kerckhoffs' principle* enunciated by Auguste Kerckhoffs von Nieuwenhof [106] that expressly requires that encryption and decryption algorithms are part of the public knowledge.

We assume that cryptograms may be generated with *symmetric-key* or *asymmetric-key* systems. A 4-tuple $(N, K, \_^{-1} : K \to K, M)$ is a *crypto-system* if,

- N is the set of plain text;
- K is the set of possible keys;
- $\_^{-1}$, given an encryption key, returns the corresponding decryption key and viceversa. We assume that $(\_^{-1})^{-1}$ is the identity function and that no further restrictions are imposed on $\_^{-1}$.
- M is the set of messages; it is obtained by closing N under encryption and pairing functions.

Hereafter, $\{m\}_k$ denotes the cryptogram obtained by encrypting message m with key k, while m, n denotes the pair made of messages m and n. As usual, we assume that keys have no structure, i.e. they are simply names.

Symmetric Key Cryptography These crypto-systems, also known under the name of *private key crypto-systems*, are characterized by the fact that encryption and decryption keys are equal (namely, $\_^{-1}$ is the identity function). Two principals say A and B, can encrypt/decrypt data if they share a key k. It is usually assumed that k is known only by A and B and other principals may acquire k only if A or B explicitly send it.

Note that a symmetric crypto-system works in both directions in the communication between A and B. Furthermore, encryption and decryption processes are very efficient. The principal drawbacks of symmetric key cryptography are:

- A and B are connected by a network and may be at a very long distance. Establishing the shared key k would require either that A meets B or that a secure key exchange protocol is adopted;

- if we consider n principals, the number of shared key that must be constructed is $\frac{n(n-1)}{2}$. Furthermore, at each communication the receiver does not know who is the actual sender and must check, in the worst case, a number of key that is quadratic in the number of principals;

- all keys must be kept secret; when a key is leaked, the involved principals must build a new key.

The most famous symmetric key algorithm is the *Data Encryption Standard* (DES) [137].

Asymmetric Key Cryptography These crypto-systems, also known as *public key crypto-systems*, are characterized by the fact that encryption and decryption keys are different. Public key cryptography has been introduced by Diffie and Hellman [63, 64] and the most famous public key crypto-system is RSA [156]. These crypto-systems are based on *public* and *private* key pairs. Each principal A has a private key A^- and a different public key A^+. It is usually assumed that, for each principal A, its public key A^+ is publicly available and may be used by any other principal to encrypt messages intended for A. Such cryptograms may be decrypted only using the private key A^- that is known only by A. Using a public key crypto-system only the intended recipient can read the messages encrypted with his/her public key, anyone can encrypt using the public key of a principal and there is no need of secure channel to communicate the public key; any principal generates his/her own public-private pair key and publish the public key. This algorithm of

key generation allows two principals to generate the same pair of keys. Anyway, for the perfect encryption hypothesis, we assume that this event is not possible[1].

The main drawbacks are:

- private keys must be maintained secret;
- when A asks the public key for a principal B to a key-server, it must be sure that the key has been originated from B;
- it must not be possible to deduce the secret key from the public one;
- encryption and decryption of messages is computationally expensive.

Public key crypto-system may be also used for implementing *digital signature* systems. Indeed, A can use A^- to encrypt messages. Such messages can be decrypted using A^+ by any other principal that, in this way, it is sure that the messages has been originated by A. A symmetric key is denoted by k while a pair of public/private key for a principal A are denoted by A^+ and A^-, respectively. Finally, we let λ to range over K and denote with λ^- its corresponding inverse key. More precisely, $\lambda^- = \lambda$ if λ is a symmetric key, while $\lambda^- = A^-$ if $\lambda = A^+$ and $\lambda^- = A^+$ if $\lambda = A^-$. We can now define the syntax of *messages*:

For the reasons reported, both public and private crypto-systems are used. In fact, a public key protocol may be used to exchange a session key and then, the session key may be used an encryption/decryption key of a private key protocol that protect interactions of two principal from intruders.

We end this section by saying that many other kinds of cryptographic systems may be found in literature; for instance, *one-way hash function*, *key agreement*, etc. Here we have only detailed public and private key crypto-systems simply to give an account of what are the aims of cryptography and to fix some notational conventions.

8.2 Protocol specification

A *security protocol* may be naively thought of as a finite sequence of messages between two or more participants. There is a great variety of specifications mechanisms of protocols and their properties. Some protocols are informally specified mixing natural language and ad hoc notation (for instance, SSL [84], SSH [176], IKE [92] are specified in this style). Other specification mechanisms are more formal and adopt precise mathematical statements sometimes expressed in formal calculi. In the remaining part of this section, we will essentially review the notions introduced in [1].

[1]The probability of generating the same pair is comparable to the probability of guessing a secret key.

Protocol specifications are usually presented by a list of message exchanges. The syntax of such communications is the following:

$$(n) \quad A \rightarrow B : \quad m.$$

The intended meaning of this notation should be:

> the protocol designer specified that the n-th step consists of the principal A sending message m and of m being received by B.

A protocol specifies various rôles: it is a diffused convention that A and B are used to represent two "normal" principals of the protocol, in particular, A is the *initiator* and B is the *responder*, i.e. the sender and the receiver of the first message, respectively; S is reserved to denote a third-part (usually) trusted server while I denotes the intruder. In some case, I_A or $I(A)$ specifies that the intruder I is acting in the protocol "imitating" A.

A sequence of message exchanges is not a complete specification. For example,

- It is not specified that only B must receive message m (usually this is desirable in security protocols).
- A specification should define, for each principal, the set of initially known data and how this set evolves during the protocol execution. Furthermore, it should be specified which are the freshly generated data.
- Tests on messages in a security protocol are important. If during a check a principal can or cannot get aware of the type of messages is important because, *type* flaws are possible if the shape of the message cannot be recognized or if a principal implicitly assumes that the messages (s)he is reading from the network have a given form.
- The specification of the number of messages, may induce an order in the communications of the protocol that is not intended by the specifier. This is the situation when simple protocols are considered, but protocols that allow multiple messages being sent simultaneously also exist.
- The assumption that a protocol may have multiple simultaneous runs and that principals may play different rôles in different runs is normal folklore.

It is quite evident that the robustness of a system relies on the security of various levels of its architecture (the cryptographic level, the protocol level, the application level considered with respect to security policies) and their relationships. At cryptographic level, it is usual to assume the so-called "perfect encryption hypothesis" stating that a cryptogram can be decrypted only using its decryption key and that a key cannot be guessed, no matter how much information is possessed. Even if such hypothesis is not completely realistic because it does not takes into account

the so called *cryptoanalysis attacks*[2], it is possible to have keys that cannot be deduced in polynomial time by "realistic" intruders, i.e. intruders that have a given computational capacity.

Example 8.2.1 *As an example of protocol we give the informal specification of the Needham-Shroeder public key protocol. For simplicity we omit the initial steps of the protocol. The protocol authentication phase is specified as follows:*

$$\begin{array}{llll} (1) & A & \rightarrow & B: \ \{na, A\}_{B^+} \\ (2) & B & \rightarrow & A: \ \{na, nb\}_{A^+} \\ (3) & A & \rightarrow & B: \ \{nb\}_{B^+}. \end{array}$$

Message (1) *prescribes that A generates a name na that is supposed to be a newly generated name that is intended to be used only for a session of the protocol and cannot be "guessed" by any other participant of the protocol: Such names are referred as* nonces. *Principal A encrypts nonce na together with its own name under the public key of B and sends the cryptogram to B. In message* (2)*, B replies with the couple of nonces na, nb encrypted with the public key of A, where nb is a new nonce generated by B. Finally, A confirms of being the "right" partner by sending back to B the nonce nb encrypted with the public key of B.*

8.3 Security properties

Many security properties can be stated for a given protocol. For instance, beside the traditional *secrecy*, *authentication* and *integrity properties*, e-commerce protocols try to achieve *fairness*, *non-repudiation*, *anonymity*, *deniability*, etc. Formalization of these properties is difficult: many different definitions have been given for each one of them. We mainly focus on secrecy, authentication and integrity that probably are the most fundamental properties that must be granted by a protocol. Intuitively, a protocol guarantees secrecy over a set of data if it is not possible that an intruder will get such data (during an execution of the protocol). A protocol guarantees authentication of a user B to a user A if, after running the protocol, A may *safely* assume that B was involved in the protocol run. Secrecy and authentication are related. Indeed, before sending secrets to anyone, a user should be sure that (s)he is "speaking" to the intended partner. Vice versa, authentication is normally achieved through exchange of some data that the protocol ensure to be created by the intended partner and nobody else. Integrity aims at guaranteeing that, once a datum has been provided, it cannot be altered by any intruder.

Security properties are not uniformly stated and defined. Generally, the kind of a property and its adequacy depends on applications. For instance, in an electronic

[2] A cryptoanalytic attacks are performed by collecting a great number of cryptograms and then analyzing them for deducing cryptographic keys.

commerce application properties like *fairness* or *non-repudiation* are requested together with secrecy, integrity and authentication; on the other hand, a mandatory feature of a voting system is, for example, *anonymity*. It is evident that a protocol can guarantee some properties but not other ones; indeed, some properties are even "incompatible" (for instance, authentication with respect to anonymity).

If secret information must be communicated in an untrusted environment, the protocol must ensure at least that possible eavesdroppers cannot understand them (secrecy), that the partner is really the intended one and that the message are really forged by the "right" subject (message authenticity). In a certain sense, secrecy, authentication and integrity are the "elementary" properties that a protocol aims at guaranteeing. Indeed, in a distributed environment the partners of a communication establish a connection over public and untrusted network and before allowing access to sensible resources or services, they must be granted of the partner identity, that resources are authentic and data are not altered during communication.

In the following sections, we will not give rigorous definitions of the security properties that we considered because we have not introduced any formal protocol specification language yet and, therefore, we are not able to formally define concepts like "the run of a protocol", "the end of a protocol run" or what exactly are "multiple protocol sessions". For the purpose of this section, however, an intuitive comprehension of these concepts suffices.

Let

$$\begin{array}{ccc} (1) & A_1 \rightarrow B_1 : & m_1. \\ \vdots & \vdots & \vdots \\ (n) & A_n \rightarrow B_n : & m_n \end{array}$$

be a protocol specification. It represents the declarative part of the protocol and it is the starting point for defining other concepts like:

- a run, or a session, of a protocol may be thought of as the execution of an implementation of the protocol specification;
- a run completes when A_n sends the message m_n. We remark that this does not mean that B_n receives m_n! Anyway, this is an intuitively "correct" termination condition if we think that network communications are asynchronous;
- many sessions of a protocol may be concurrently executed and a principal may be involved in more than one session with different partners and assuming different rôles;
- we assume that it is possible to linearly order different runs of a protocol; given two runs, this allows us to decide which is the first that has been activated.

Secrecy This property is an important feature of security protocols: Secrecy[3] must be guaranteed on *sensible* data like secrete keys, principal identities, or nonces.

[3]Other terms for denoting secrecy are 'privacy' and 'confidentiality'.

Many security properties are based on sharing of secrets. For instance, a principal authenticates itself to a server by providing a secret (a nonce or a password).

Informally, secrecy may be expressed as

> a protocol provides secrecy on one of its parameters, say x, if an intruder can not obtain informations by analyzing the messages assigned to x during the execution of the protocol.

Generally, cryptography is not enough to guarantee secrecy, in fact most precautions are required. For instance, if a one-bit message b must be communicated between two principals [1] cryptography is not enough. If b is encrypted with a key k and sent more times on public channels, an intruder could deduce its value by comparing the cryptograms.

There are at least two possible way to specify secrecy of a protocol. The first says that a protocol guarantees secrecy on one of its parameters x if it does not leak *any* information about x, or, equivalently, if the value of x does not interfere with the behaviour of the protocol that the environment can observe. If we use a process calculus to implement protocols, such property may be rephrased saying that a process (i.e. a protocol) $P(x)$ guarantees secrecy on x if, for all possible value m, m', $P(m)$ *is equivalent*[4] to $P(m')$. The second criterion defines secrecy as a behavioural property and can be traced back to the work of Dolev and Yao [65]. According to that definition, a process preserves the secrecy of a piece of data m if it never sends m or anything that would allow the computation of m in clear on public channels.

We remark, as done in [171], that the given intuitions of secrecy are very different each other. The former concerns a free variable x, while the latter is related to a closed process and a term with no free variables. Furthermore, the first definition rules out implicit information flows, while the second takes them into account.

In some sense, secrecy deals with *information flow*. Indeed, generally speaking, secrecy properties try to limit the "deductions" of intruders on data exchanged in a protocol; therefore, secrecy may be viewed as the ability of a protocol to restrict flows of information from "normal" principals to intruders. Roughly speaking, secrecy aims at guaranteeing that if a subject acquires some informations then it is authorized to obtain them.

Authentication The formalization of this concept seems very difficult to be stated. The intuitive notion of authentication may be expressed as

> a participant should become sure of the identity of its partners or of the principals that forge messages that (s)he receives.

In this sense, if a protocol aims at providing some form of authentication to a principal A of a principal B then, after having run the protocol (*apparently*) with B, A can deduce something on the state of B. For example,

[4]The equivalence relation may be a testing equivalence [60].

- A could deduce that B has recently been alive,
- B can deduce that he had recently run the protocol with A,
- A could think that B believes that he was running the protocol with A,
- both A and B can assume that they run the same session of the protocol A as the initiator and B as a responder.

All the previous conditions correspond to different authentication properties and should be formally defined in the design of a protocol. For an extended discussion on this topic we refer the reader to [115]. However, protocols specifications rarely provide a rigorous definition of the meant authentication property. This is problematic because a user may believe that a protocol satisfies a stronger condition than the one intended by the designer.

Example 8.3.1 *The Needham-Schroeder protocol, introduced in Example 8.2.1, has been designed to achieve reciprocal authentication of A and B by exchanging nonces na and nb. Hence, when B receives the last message, it assumes that A has been its partner in a protocol session where nonce nb was generated in response to the first message $\{na, A\}_{B^+}$ that B had interpreted as "A created the nonce na to gain authentication from me".*

The Needham-Schroeder protocol was supposed to guarantee the authentication property of Example 8.3.1 for many years. Under the perfect encryption assumption, A and B can make many deduction on the reciprocal intentions. For instance, A can assume that only B can decrypt cryptogram $\{na, A\}_{B^+}$ because only B owns the private key B^-. Similarly, when B receives back its nonce nb, he can infer that only A could know nb because the only way for a principal different from B to know it is to decrypt $\{na, nb\}_{A^+}$ and this can be done only by A. However, it is well known that Needham-Schroeder protocol is flawed.

We remark that authentication can also involve "recentness". The deductions that a principal A can do on the state of another principal B not always are relative to a current or recent state of B and may even regard other protocol sessions. "Recentness" is not easy to formalize in many protocol specifications because they do not provide any mechanism to specify the passage of time.

Integrity This property deals with the detection and correction of "modification" (i.e. insertion, deletion, replay, etc.) of transmitted data including both intentional or malicious manipulations [89]. The intuitive notion of integrity may be expressed as

> a participant should be sure that data (s)he gets during the execution of a protocol has not altered in a "non-prescribed" manner.

In this sense, integrity is the dual of secrecy (which deals with information that may be "read" by participants of the protocol) because integrity concerns the capabilities of altering the information exchanged using a protocol.

In a distributed setting where attackers have the power of "destroying" messages sent along public channels, integrity may not be completely achieved because some messages may be captured and discarded by an attacker. Anyway, such kind of attacks are similar to a network fault and usually are not "dangerous"; the only thing that an attacker gains with this kind of attacks is that the run of a protocol is interrupted and a new attempt is done by normal principals.

Cryptography is a necessary prerequisite for integrity but it is not enough. Substitution of messages encrypted by key previously sent on public channels or replay attacks must be considered. In order to detect this kind of attacks, some precautions must be taken; for example, using timestamps or nonces may allow to reveal if a message is a new message or not.

From the considerations reported, integrity appears as an indispensable property that is used also to ensure other security properties, but difficult to capture and analyze using formal methods.

Chapter 9

A Formal Framework for Security

Abstract

The sections contained in this chapter briefly outline the well known Dolev-Yao model of intruder for cryptographic protocols. We follow a quite standard approach that models intruder's capabilities in terms of a relation $\bowtie$ that says if a message can be derived from a give "knowledge". As a minor contribution, we show decidability of $\bowtie$ in the case of asymmetric keys. As far as we know, a natural deduction style proof have been given in [48] in the case of symmetric keys. This proof does not scale to asymmetric key cryptography because encryption and decryption operations cannot be seen as correspondent introduction and elimination.

We also introduce and discuss cIP a process calculus that can be thought of as an extension of π-calculus to cryptography. Its adequacy and differences to other similar proposal are also discussed.

Finally, we define a simple logic for expressing security protocols in terms of relationships among the variable of the cIP processes.

Contents

9.1 Intruder model

A formal framework for protocol analysis must declare which assumptions are made on the intruder. Indeed, different models permits one to model different classes of attacks. For instance, if intruders can be *passive* or *active* makes difference. In the first case, an intruder can only listen messages sent by regular principals, but cannot sent anything. On the other hand, active intruders also can forge messages and send them to other participants. Intuitively, it is clear that active intruders are more "powerful" than passive ones and can also "simulate" them. Indeed, they can intercept a message and forward to regular receivers.

We remark that modeling a form of intrusion is intimately related to detailing a system at some level of abstraction. For instance, if we are interested in analyzing protocols that can be attacked by intruders able in guessing keys, we must describe how keys are represented, establish a random distribution over the space of keys and so on. On the other hand, if we are not interested in such kind of attacks, we can simply represent keys as atomic names.

A well-known and commonly adopted model is the Dolev-Yao model [65]. This model describes an active intruder that can

- receive and store any transmitted message;
- hinder a message;
- decompose message into parts;
- forge messages using known data.

In this model, the only limitation for intruders is imposed by the perfect encryption assumption: Intruders cannot guess keys that they do not explicitly own. This assumption has consequences on the activity an intruder can do, e.g. decomposing messages, forging messages and so on. For instance, a cryptogram can be disclosed by an intruder only if he has the decryption key and, dually, an intruder can forge a cryptogram only if it is a known datum or the encryption key is owned.

It is diffused costume to assume that intruders also can have data not expressly generated by regular principals. This amount to assume that intruders can "remember" data exchanged in previous run of the protocols.

Observe that the Dolev-Yao model characterizes an intruder it terms of the knowledge he has of the data exchanged by principals. In particular, an intruder can record all exchanged messages and use them later to fake principals or to extract data that must be kept secret.

Since an intruder *á la* Dolev-Yao can intercept any communication, without loss of generality, we can consider the execution environment as the "adversary". Moreover, the environment can record all sent messages and manipulate them when a principal is waiting for some data (see Figure 9.1).

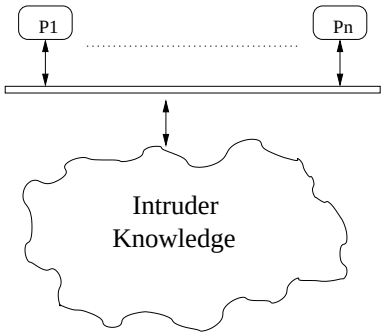


Figure 9.1: A graphical representation of the Dolev-Yao intruder

9.2 Formalizing the Intruder Model

Following the Dolev-Yao model [66], principal communications can be intercepted by any other principal in the environment and in particular by malicious ones. Hence, intruders are characterized by the knowledge they acquire. Intruders can also be equipped with a non-empty initial knowledge representing what they learned in previous protocols runs. This permits us to study the robustness of a protocol with respect to the power of different intruders, e.g. from the one which has no previous knowledge about the protocol, to those which know part of or all the secret keys of the principals.

Let N_o be the set of nonces and let N_p be the set of principal names; N_p is ranged over by A, B, S; we assume that $N_o \cap N_p = \emptyset$ and denote their union with N. Let K be a set of keys; we assume that K contains both symmetric and asymmetric keys and that $K \cap N = \emptyset$.

Definition 9.2.1 (Messages) *A* message *is a term derived as follows:*

$$M \quad ::= \quad N \quad | \quad K \quad | \quad M, M \quad | \quad \{M\}_M.$$

We let m, n, ... to range over M, while λ ranges over K.

A message may be a name (i.e. a nonce or a principal name), a key (symmetric or not), the pairing of two messages or the encryption of a message.

Notation 9.2.1 *Tuples of messages are generated by the production M, M that, differently from the usual notations, are not delimited with brackets. Essentially, we want to consider messages like m_1, m_2, m_3 as triples which can be matched only by patterns that are triples, whereas, we shall write $(m_1, m_2), m_3$ for indicating a pair whose first component is the tuple m_1, m_2 (and similarly for $m_1, (m_2, m_3)$). Notice that in this manner we resolve the ambiguity of matching a pattern p_1, p_2 that cannot match m_1, m_2, m_3.*

As usual, we assume that only atomic keys, namely elements of K, can be used to encrypt messages. Therefore, Definition 9.2.1 is an over-specification of the set of

messages (because of the last production). However, it is necessary because "wrong" messages can be generated during principal communications (see Section 9.4).

Definition 9.2.2 formalizes the capabilities that the Dolev-Yao model ascribes to intruders (as pointed out in Section 9.1). Indeed, κ may be thought of as the intruder's knowledge and each inference rule describes the possible manipulation that the intruder may perform on data contained in κ.

Definition 9.2.2 (Intruder capabilities) *Given $\kappa \subseteq M$ and $m \in M$, we say that m* can be derived from *κ, and write $\kappa \bowtie m$, if $\kappa \bowtie m$ is obtained with a finite proof by the following inference rules:*

$$\frac{m \in \kappa}{\kappa \bowtie m}\,(\in) \qquad \frac{\kappa \bowtie m \quad \kappa \bowtie n}{\kappa \bowtie m, n}\,(,) \qquad \frac{\kappa \bowtie m \quad \kappa \bowtie \lambda}{\kappa \bowtie \{m\}_\lambda}\,(\{\}) \qquad \frac{\kappa \bowtie m, n}{\kappa \bowtie m}\,(+_1) \qquad \frac{\kappa \bowtie m, n}{\kappa \bowtie n}\,(+_2) \qquad \frac{\kappa \bowtie \{m\}_\lambda \quad \kappa \bowtie \lambda^-}{\kappa \bowtie m}\,(\}\{)$$

Let us remark that rules $(\in)$, $(,)$ and $(\{\})$ permits the intruder to forge messages from messages in its knowledge κ. For instance, rule $(\{\})$ states that the intruder can encrypt any message m deducible from κ with any key λ, provided that also λ is deducible from κ. Rules $(+_1)$, $(+_2)$ and $(\}\{)$ give the "dual" capabilities. Indeed, they allow the intruder to decompose a message into its constituent. Note how rule $(\}\{)$ states that a cryptogram can be decrypted only if κ contains enough information to deduce the corresponding decryption key. This is coherent with the perfect encryption hypothesis which prescribes that keys cannot be guessed.

Example 9.2.1 *We show how a message can be deduced from a set of messages using rules in Definition 9.2.2. Let κ be the set $\{k, \{\{a\}_{A^-}\}_k, A^+\}$. We prove that $\kappa \bowtie \{a\}_{A^+}$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{k, \{\{a\}_{A^-}\}_k \in \kappa}{\kappa \bowtie k, \{\{a\}_{A^-}\}_k}\,(\in)}{\kappa \bowtie \{\{a\}_{A^-}\}_k}\,(+_2) \qquad \dfrac{\dfrac{k, \{\{a\}_{A^-}\}_k \in \kappa}{\kappa \bowtie k, \{\{a\}_{A^-}\}_k}\,(\in)}{\kappa \bowtie k}\,(+_1)}{\kappa \bowtie \{a\}_{A^-}}\,(\}\{) \qquad \dfrac{A^+ \in \kappa}{\kappa \bowtie A^+}\,(\in)}{\kappa \bowtie a}\,(\}\{) \qquad \dfrac{A^+ \in \kappa}{\kappa \bowtie A^+}\,(\in)}{\kappa \bowtie \{a\}_{A^+}}\,(\{\})$$

Example 9.2.1 shows how message deduction can be conducted. We remark that the structure of the proof is quite general. Indeed, hypothesis of the proof all have the form $m \in \kappa$, while the conclusion $\kappa \bowtie \{a\}_{A^+}$, is the message we want to deduce. This suggests us that κ contains the "hypothesis" of our deduction system and the proof ends with the thesis. Definition 9.2.3 and Lemma 9.2.1 formalize this observation and will be useful for later results.

Notation 9.2.2 *In the following we indicate with* $\frac{T_1 \cdots T_i}{r}$ *a tree whose root is* r *and the sons of* r *are the subtrees* T_1,...,T_i.

Definition 9.2.3 (Proof tree) *Let* m *be a message. A* proof tree for m *is a tree, whose nodes are messages and such that:*

1. *a node* m *is a proof tree for* m;
2. *if* T_i *are proof trees for* m_i $(i = 1, 2)$ *the* $\frac{T_1 \quad T_2}{m_1, m_2}$ *is a proof tree for* m_1, m_2;
3. *if* T_1 *is a proof tree for* m *and* T_2 *is a proof tree* λ *then* $\frac{T_1 \quad T_2}{\{m\}_\lambda}$ *is a proof tree for* $\{m\}_\lambda$;
4. *if* T *is a proof tree for* m_1, m_2 *then* $\frac{T}{m_i}$ *are proof tree for* m_i, $i = 1, 2$;
5. *if* T_1 *is a proof tree for* $\{m\}_\lambda$ *and* T_2 *is a proof tree* λ^- *then* $\frac{T_1 \quad T_2}{m}$ *is a proof tree for* m.

Lemma 9.2.1 *For each* $\kappa \in \wp(M)$ *and each* $m \in M$, $\kappa \bowtie m \iff \exists T$: T *proof tree for* m *such that the root of* T *is* m *and each leaf of* T *is a message in* κ.

Proof.

($\Rightarrow$) By definition a finite proof of $\kappa \bowtie m$ exists. We reason by induction on the length of such proof. If the proof has length 1, then the only possible proof is the application of rule ($\in$) of Definition 9.2.2. Then m trivially is a proof tree for m ($m \in \kappa$ for the premise of the ($\in$) rule). If the proof for $\kappa \bowtie m$ has length $h + 1$ then the last applied rule should be one of ($\in$), ($,$), ($\{\}$), ($+_1$), ($+_2$), ($\}\{$). For each rule, the premises have proofs with length lesser than $h + 1$ that, by inductive hypothesis, correspond to proof trees with the properties described in the statement. Then we can apply one of the rules in Definition 9.2.3 to obtain the searched proof tree for m.

($\Leftarrow$) This direction is proved by induction on the structure of T and by case analysis on the root of T. If $T = m$ then, by hypothesis, $m \in \kappa$ and the rule ($\in$) can be applied to prove $\kappa \bowtie m$. if $T = \frac{T_1 \quad T_2}{m_1, m_2}$ is a proof tree for m_1, m_2, then, by definition, T_i is proof tree for m_i and, by induction hypothesis, $\kappa \bowtie m_i$ and κ contains the leave of T_1 and T_2, then T is a proof tree for m whose leave are all in κ. Other cases are analogous.

□

9.3 Decidability of $\bowtie$

In [48] it has been proved that $\bowtie$ is a decidable relation as far as symmetric cryptographic systems are considered. The proof relies on the fact that, for symmetric key cryptography, rules in Definition 9.2.2 behave as natural deduction rules for logical inference. This proof does not scale to asymmetric key cryptography. The problem is that cryptogram introduction rule ($\{\}$) does have a symmetric elimination rule only as far as symmetric keys are dealt, where condition $\kappa \bowtie \lambda^-$ is equivalent to $\kappa \bowtie \lambda$, hence, ($\{\}$) is the introduction rule for cryptograms and ($\}\{$) is the corresponding elimination rule. Indeed, the case for asymmetric keys is different because the decryption key must be deduced and it is different for the encryption key. We show that $\bowtie$ is decidable also when asymmetric-keys cryptography is considered.

The idea of the proof is that the derivation of a message m from a set κ can be divided in two parts: first, all sub-terms deducible from κ are computed (by applying, in all possible way, message destructors, namely, decryption and decoupling); finally, by applying message constructors (encryption and pairing), we try to derive m.

Definition 9.3.1 (Message decomposition) *We say that κ can be* simplified *if one of the inference rules below can be applied*

$$\frac{m, n \in \kappa}{\kappa \stackrel{m,n}{\rightharpoonup} \kappa \setminus m, n \cup m \cup n} \qquad \frac{\{m\}_\lambda \in \kappa \quad \lambda^- \in \kappa}{\kappa \stackrel{\{m\}_\lambda}{\rightharpoonup} \kappa \setminus \{m\}_\lambda \cup m}$$

Definition 9.3.1 describes a labelled transition system whose states are set of messages and whose labels are messages. As usual, we write $\kappa \stackrel{m_1 \ldots m_h}{\rightharpoonup} \kappa_{h+1}$ if κ_1,...,κ_{h+1} exist such that $\kappa \stackrel{m_1}{\rightharpoonup} \kappa_1 \stackrel{m_2}{\rightharpoonup} \ldots \stackrel{m_{h-1}}{\rightharpoonup} \kappa_h \stackrel{m_h}{\rightharpoonup} \kappa_{h+1}$. By convention, when $h = 0$, $\kappa \stackrel{m_1 \ldots m_h}{\rightharpoonup} \kappa_{h+1}$ is equivalent to $\kappa = \kappa_1$.

Example 9.3.1 *Let us consider the set $\kappa = \{k, \{\{a\}_{A^-}\}_k, A^+\}$ of Example 9.2.1. By applying the rules of Definition 9.3.1 we have the following transitions:*

$$\kappa \stackrel{k,\{\{a\}_{A^-}\}_k}{\rightharpoonup} \{k, \{\{a\}_{A^-}\}_k, A^+\} \stackrel{\{\{a\}_{A^-}\}_k}{\rightharpoonup} \{k, \{a\}_{A^-}, A^+\} \stackrel{\{a\}_{A^-}}{\rightharpoonup} \{k, a, A^+\}$$

Note that the union of the final set and the label of all transitions is the set of all sub-messages that can be deduced by κ.

Definition 9.3.1 describes two rules that simplifies a set of messages by removing those messages that can be reconstructed with a "simpler" set of messages. Here, "simpler" may be formally defined. Let us consider the following definition.

Definition 9.3.2 (Message size) *Given a message $m \in M$, the* size of m, $\#m$, *is defined by induction on the structure of m as follows:*

$$\#m \;=\; \begin{cases} 1, & \text{if } \; m \in N \cup K \\ 1 + \#m_1 + \#m_2, & \textit{if } \; m = m_1, m_2 \\ 1 + \#n, & \textit{if } \; m = \{n\}_\lambda. \end{cases}$$

Given $\kappa \in \wp_{fin}(M)$, *the* size of κ, $\#\kappa$, *is defined as* $\#\kappa = \sum_{m \in \kappa} \#m$. *We let* $\#\emptyset = 0$.

As usual, the complexity of a message is proportional to the number of productions necessary to derive it from the productions for M (see Definition 9.2.1). The following lemma states that $\rightharpoonup$ reduces the complexity of a finite set of messages.

Lemma 9.3.1 *Given* $\kappa \in \wp_{fin}(M)$ *and* $\bar{m} \in M$ *, if* $\kappa \stackrel{\bar{m}}{\rightharpoonup} \kappa'$, *then* $\#\kappa' < \#\kappa$.

Proof. There are two possible cases:

$\boxed{\bar{m} = m, n}$ By definition of $\rightharpoonup$ we have

$$\#\kappa' = \sum_{m' \in \kappa \setminus \bar{m}} \#m' + \#m + \#n = \sum_{m' \in \kappa} \#m' - \#\bar{m} + \#m + \#n \quad (9.1)$$

Finally, from equality (9.1) and $\#\bar{m} = 1 + \#m + \#n$ we can conclude that $\#\kappa' = \#\kappa - 1 < \#\kappa$.

$\boxed{\bar{m} = \{n\}_\lambda}$ By mimicking the previous case, we have:

$$\#\kappa' = \sum_{m' \in \kappa \setminus \bar{m}} \#m' + \#n = \sum_{m' \in \kappa} \#m' - \#\bar{m} + \#n \quad (9.2)$$

and, as before, we conclude $\#\kappa' = \#\kappa - 1 < \#\kappa$.

This concludes the proof. □

Lemma 9.3.1 states that $\rightharpoonup$ transforms a set of messages with another set having a strictly lower size.

Another useful observation is that the application of $\stackrel{m}{\rightharpoonup}$ to a set does not removes messages different from m:

Lemma 9.3.2 *Given* $\kappa \in \wp(M)$. *let* $\bar{m} \in \kappa$ *and* $\kappa \stackrel{m}{\rightharpoonup} \kappa'$ *be such that* $\bar{m} \neq m$, *then* $\bar{m} \in \kappa'$.

Proof. If $\kappa \stackrel{m}{\rightharpoonup} \kappa'$ then $\kappa' = \kappa \setminus m \cup V$, where V depends on m. Hence, the only message removed from κ is $m \neq \bar{m}$. □

Corollary 9.3.2.1 *Given a set* $\kappa \in \wp(M)$ *such that* $\kappa \stackrel{m}{\rightharpoonup} \kappa'$ *for a given message* m. *Then for any* $\bar{m} \in \kappa$ *such that* $\#\bar{m} = 1$, $\bar{m} \in \kappa'$.

Proof. For each transition $\kappa \stackrel{m}{\rightharpoonup} \kappa'$, $\#m > 1$, hence $\bar{m} \neq m$. □

Previous corollary implies that $\rightharpoonup$ never removes keys or names.

Interestingly, $\rightharpoonup$ is confluent on finite set of messages, as stated below:

Proposition 9.3.1 (Confluence of $\rightharpoonup$) *For any $\kappa \in \wp_{fin}(M)$, if $\kappa \overset{m_1}{\rightharpoonup} \kappa_1$ and $\kappa \overset{m_2}{\rightharpoonup} \kappa_2$, then exist $\overline{\kappa}$, $m_1, ..., m_h$ and $m'_1, ..., m'_{h'}$ such that*

$$\kappa_1 \overset{m_1 ... m_h}{\rightharpoonup} \overline{\kappa} \qquad \wedge \qquad \kappa_2 \overset{m'_1 ... m'_{h'}}{\rightharpoonup} \overline{\kappa}$$

Proof. Let $\kappa \overset{m_i}{\rightharpoonup} \kappa \setminus m_i \cup V_i$, $i = 1, 2$ (where V_i depends on m_i) be two distinct transitions from κ. Lemma 9.3.2 implies that the following transitions can be deduced from κ:

$$\begin{array}{ccc} & \kappa \setminus m_1 \cup V_1 \xrightarrow{m_2} \kappa_1 = (\kappa \setminus m_1 \cup V_1) \setminus m_2 \cup V_2 & \\ \overset{m_1}{\nearrow} & & \\ \kappa & & \\ \underset{m_2}{\searrow} & & \\ & \kappa \setminus m_2 \cup V_2 \xrightarrow{m_1} \kappa_2 = (\kappa \setminus m_2 \cup V_2) \setminus m_1 \cup V_1. & \end{array} \tag{9.3}$$

The proof proceeds by case analysis on m_1 and m_2.

$\boxed{m_1 = p_1, q_1,\ m_2 = p_2, q_2}$ In this case, $V_i = \{p_i, q_i\}$ for $i = 1, 2$.

We distinguish the case where one of p_1 or q_1 is equal to m_2 from the case where both are different from m_2. Under the assumption that $p_1 = p_2, q_2$, we can conclude two facts:

$$\kappa_1 = (\kappa \setminus m_1 \cup q_1) \setminus m_2 \cup p_2 \cup q_2$$

which follows from (9.3), and

$$\begin{aligned} \kappa_2 \overset{m_2}{\rightharpoonup} \kappa'_2 &= ((\kappa \setminus m_2 \cup p_2 \cup q_2) \setminus m_1 \cup q_1) \setminus m_2 \cup p_2 \cup q_2 \\ &= (\kappa \setminus m_2 \cup p_2 \cup q_2 \cup q_1) \setminus \{m_1, m_2\} \cup p_2 \cup q_2 \qquad (9.4) \\ &= (\kappa \setminus m_2 \cup q_1) \setminus \{m_1, m_2\} \cup p_2 \cup q_2 \qquad (9.5) \\ &= (\kappa \setminus m_2 \cup q_1) \setminus m_1 \cup p_2 \cup q_2 \qquad (9.6) \end{aligned}$$

Previous equalities are derived by observing that both p_2 and q_2 are different from m_1 because their size is strictly less than $\#m_1$ (equality (9.4)) and by set theoretical arguments (equalities (9.5) and (9.6)). Last equality implies that $\kappa'_2 = \kappa_1$. If $q_1 = p_2, q_2$ the proof is analogous.

If $p_1, q_1 \neq p_2, q_2$, then we can assume that both p_2 and q_2 differ from m_1 (otherwise, we proceed analogously to the previous case). In this case the following equalities hold:

$$\kappa_1 = \kappa \setminus \{m_1, m_2\} \cup \{p_1, q_1, p_2, p_2\} \qquad \wedge \qquad \kappa_2 = \kappa \setminus \{m_2, m_1\} \cup \{p_2, q_2, p_1, p_1\}.$$

$\boxed{m_1 = \{n_1\}_{\lambda_1},\ m_2 = \{n_2\}_{\lambda_2}}$ In this case $V_i = \{n_i\}$, where $i = 1, 2$.

Let us assume that $n_1 = \{n_2\}_{\lambda_2}$. Then we have $\kappa_1 = (\kappa \setminus \{m_1, m_2\}) \cup n_2$ and

$$\kappa_2 \overset{m_2}{\rightharpoonup} \kappa'_2 = (((\kappa \setminus m_2) \cup n_2) \setminus m_1) \setminus m_2 \cup n_2. \tag{9.7}$$

In (9.7), $\kappa_2 \overset{m_2}{\rightharpoonup} \kappa'_2$ because of Corollary 9.3.2.1, then $\kappa_1 = \kappa'_2$. If $m_2 = m_1$ we proceed in an analogous manner, while if $n_1 \neq m_2$ and $n_2 \neq m_1$ we simply have

$$\kappa_1 = \kappa \setminus m_1 \setminus m_2 \cup n_1 \cup n_2 \qquad \wedge \qquad \kappa_2 = \kappa \setminus m_2 \setminus m_1 \cup n_2 \cup n_1$$

that trivially implies $\kappa_1 = \kappa_2$.

$\boxed{m_1 = p, q,\ m_2 = \{n\}_\lambda}$ In this case $V_1 = \{p, q\}$ and $V_2 = \{n\}$. If $n = p, q$, then $p \neq m_2 \wedge q \neq m_2$ because $\#m_2$ is strictly greater than $\#p$ and than $\#q$. Hence,

$$\kappa_2 \overset{m_1}{\rightharpoonup} \kappa_2' = (\kappa \setminus m_1 \setminus m_2 \cup n \cup p \cup q) \setminus m_1 \cup p \cup q = \kappa \setminus m_2 \setminus m_1 \cup p \cup q.$$

On the other hand $\kappa_1 = (\kappa \setminus m_1 \setminus m_2 \cup p \cup q) \setminus m_1 \cup p \cup q$, which, by set theoretic reasoning, implies that $\kappa_1 = \kappa \setminus m_2 \setminus m_1 \cup p \cup q$ that completes the proof.
Finally, if $n \neq p, q$ then $\kappa_2 = \kappa \setminus \{m_1, m_2\} \cup \{p, q, n\}$. The proof proceeds by case analysis on p.

$p = \{n\}_\lambda$: We have $\kappa_2 \overset{m_2}{\rightharpoonup} (\kappa \setminus m_1 \setminus m_2 \cup p \cup q \cup n) \setminus m_2 \cup n = \kappa \setminus m_1 \setminus m_2 \cup q \cup n$. The proof ends by observing that $\kappa_1 = (\kappa \setminus m_1 \cup p \cup q) \setminus m_2 \cup n = \kappa \setminus m_1 \setminus m_2 \cup q \cup n$.

$p \neq \{n\}_\lambda$: In this case we can assume that $q \neq m_2$, otherwise we can proceed as in the previous case. The proof trivially follows from (9.3) by observing that, under those assumptions, set difference and set union commute.

□

Confluence of $\rightharpoonup$ is important because it allows to "minimize" the complexity of finite sets.

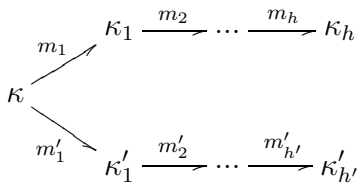
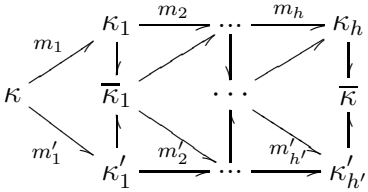
Proposition 9.3.2 *Given a finite set of messages κ, a unique set $\overline{\kappa} \in \wp_{fin}(M)$ exists such that $\kappa \overset{m_1 \ldots m_h}{\rightharpoonup} \overline{\kappa}$ for some $m_1, .., m_h \in M$ and $\overline{\kappa} \overset{m}{\not\rightharpoonup}$ for any $m \in M$.*

Proof. We first prove existence of $\overline{\kappa}$ by induction on $\#\kappa$.

$\boxed{\#\kappa = 0}$ This case is possible only when $\kappa = \emptyset$; then no rule of Definition 9.3.1 can be applied to κ, therefore, $\overline{\kappa} = \emptyset$.

$\boxed{\#\kappa = h + 1}$ If no rule can be applied, $\overline{\kappa} = \kappa$; otherwise $\kappa \overset{m}{\rightharpoonup} \kappa'$ and, by Lemma 9.3.1, $\#\kappa' < \#\kappa$. Then, by inductive hypothesis, $\kappa' \overset{m_1 \ldots m_u}{\rightharpoonup} \overline{\kappa}$ and no rule can be applied to $\overline{\kappa}$. This shows that $\kappa \overset{m\, m_1 \ldots m_u}{\rightharpoonup} \overline{\kappa}$.

Uniqueness is proved *by absurdum*.

If  then  (by iterative application of the confluence). This contradicts the hypothesis that no rule can be applied to κ_h and to $\kappa_{h'}'$. □

Propositions 9.3.2 and 9.3.1 guarantees that the following function is well defined.

Definition 9.3.3 (Decomposition function) *Let* $\partial : \wp_{fin}(M) \to \wp_{fin}(M)$ *be defined as*

$$\partial(\kappa) = \begin{cases} \emptyset, & \text{if } \kappa = \emptyset, \\ \overline{\kappa} \cup m_1 \cup ... \cup m_h, & \text{if } \kappa \xrightarrow{m_1 ... m_h} \overline{\kappa} \wedge \forall m \in M. \overline{\kappa} \not\xrightarrow{m} . \end{cases}$$

Proposition 9.3.3 *Function* ∂ *is well-defined and computable.*

Proof. By inspecting the proof of Proposition 9.3.1, we can observe that the application of a rule preserves applications of other rules. Moreover, Proposition 9.3.2 guarantees uniqueness of $\overline{\kappa}$.

Computability of ∂ derives from Lemma 9.3.1. Indeed, by applying rules in Definition 9.3.1 we will reach a set where no more rules are applicable because the size of reached sets is strictly decreasing and cannot be negative. Therefore, we can write an iterative procedure that reaches such set and collects all the labels during the rule applications. This, by Church-Turing thesis, completes the proof. □
Basically, function ∂ returns the set of all "sub-messages" that may be obtained by decomposing messages in κ according to Definition 9.2.2. Computability of ∂ is important because we will extensively use it in the semantics definition of the process calculus cIP introduced in Section 9.4.

Proposition 9.3.4 $\forall m \in M.\ m \in \partial(\kappa) \Rightarrow \kappa \bowtie m.$

Proof. We proceed by induction on $\#\kappa$.

$\#\kappa = 0$**:** We have that $\kappa = \emptyset$ and, the property trivially holds.

$\#\kappa = h + 1$**:** If no rule in Definition 9.3.1 can be applied to κ then $\partial(\kappa) = \kappa$, hence $\dfrac{m \in \kappa}{\kappa \bowtie m}$ is a finite proof for $\kappa \bowtie m$. If $\kappa \xrightarrow{\bar{m}} \kappa_1$, for a message $\bar{m} \in \kappa$, we can assume that $\partial(\kappa) = \partial(\kappa_1) \cup \bar{m}$.
If $m = \bar{m}$ then, by hypothesis, $m \in \kappa$ and, as before, a proof for $\kappa \bowtie m$ exists. Otherwise, $m \neq \bar{m}$ and $m \in \partial(\kappa_1)$ then Lemma 9.3.1 and Lemma 9.3.2 permit us to apply the inductive hypothesis and obtain $\kappa_1 \bowtie m$. By Lemma 9.2.1 a proof tree for m, T whose leave are on κ_1 exists. Observing that, by definition, $\kappa_1 = \kappa \setminus m \cup V$ where V depends on m, then we can attach to each leave m' of T in $V \setminus \kappa$ a proof tree with root m' and sons proof the trees with leave in κ built according to the transition rule $\xrightarrow{m}$. The obtained tree is a proof tree for m with leave in κ and, by Lemma 9.2.1, we obtain the thesis.

This completes the proof. □

In order to consider all possible derivation $\kappa \bowtie m$, we must also take into account message construction.

Definition 9.3.4 (Message construction) *Let κ be a subset of M. We say that κ* constructs *m, and write $\kappa \rhd m$, if, and only if, there is a proof of $k \bowtie m$ built out using only rules $(\in)$, $(,)$ and $(\{\})$*

Function ∂ returns all the submessages of a given finite subset of messages. Computing $\partial(\kappa)$ corresponds to decompose, as much as possible, elements of κ. Those submessages are the "bricks" upon which the intruder can forge new messages.

Theorem 9.3.1 $\forall \kappa \in \wp_{fin}(M). \forall m \in M.\ \kappa \bowtie m \iff \partial(\kappa) \rhd m$

Proof. $(\Leftarrow)$ The proof is given by induction on the length of the proof of $\partial(\kappa) \rhd m$. If m can be derived with rule $(\in)$, then $m \in \partial(\kappa)$ and the thesis holds by Proposition 9.3.4 .
Assume that $\partial(\kappa) \rhd m$ with a proof, $\prod$, having length $h > 1$ and that the theorem holds for all messages m' such that the proof of $\partial(\kappa) \rhd m'$ has length less than h. If $m = \{n\}_\lambda$ then the last rule applied in $\prod$ is $(\{\})$, namely:

$$\frac{\partial(\kappa) \rhd n \quad \partial(\kappa) \rhd \lambda}{\partial(\kappa) \rhd \{n\}_\lambda}$$

and, by inductive hypothesis, $\kappa \bowtie n$ and $\kappa \bowtie \lambda$ therefore, by applying $(\{\})$, we obtain $\kappa \bowtie m$.
A similar reasoning can be done when the last rule applied in $\prod$ is $(,)$.

$(\Rightarrow)$ If $\kappa \bowtie m$ then there exists a finite proof $\prod$ of it using rules in Definition 9.2.2. We proceed by induction on the length of $\prod$. If $\prod$ has length one only if the $(\in)$ rule is applied, namely, only if $m \in \kappa$. Then $m \in \partial(\kappa)$ and rule $(\in)$ can be applied to $\partial(\kappa)$ yielding the thesis. If $\prod$ has length strictly greater than 1 then one of the following cases hold:

- $m = n, n'$ and the last rule in $\prod$ is $(,)$. Then $\kappa \bowtie n$ and $\kappa \bowtie n'$ with proofs shorter than $\prod$ then, by inductive hypothesis, $\partial(\kappa) \rhd n$ and $\partial(\kappa) \rhd n'$ and, by rule $(,)$, $\partial(\kappa) \rhd m$.

- $m = \{n\}_\lambda$ and the last rule in $\prod$ is $(\{\})$ the proof is analogous to the previous case.

- If last rule in $\prod$ is $(+_1)$ then there is a message n such that $\kappa \bowtie m, n$ with a proof shorter than $\prod$. If $m, n \in \kappa$ then both m and n are in $\partial(\kappa)$ and, by applying rule $(\in)$, we obtain the thesis. Otherwise, $\kappa \bowtie m$ and $\kappa \bowtie n$ with proofs shorter than $\prod$ therefore, the thesis holds by inductive hypothesis.
  In case $(+_2)$ is the last rule in $\prod$, we proceed similarly to the case $(+_1)$.

- If the last rule in $\prod$ is $(\}\{)$ then there are a cryptogram $\{m\}_\lambda$ and a key λ^- such that $\kappa \bowtie \{m\}_\lambda$ and $\kappa \bowtie \lambda^-$ with proofs shorter that $\prod$. If $\{m\}_\lambda \notin \kappa$ then there is a proof of $\kappa \bowtie m$ shorter than $\prod$ and the inductive hypothesis gives

the thesis. Let us assume that $\{m\}_\lambda \in \kappa$. If $\lambda^- \in \kappa$ then also $\lambda^- \in \partial(\kappa)$ holds (Corollary 9.3.2.1) and, by definition of ∂, $m \in \partial(\kappa)$ because the cryptogram can be decomposed If $\lambda^- \notin \kappa$ then $\kappa \bowtie \lambda^-$ with a proof shorter than $\prod$. Because there is no rule in Definition 9.2.2 for "constructing" keys, only one of $(+_1)$, $(+_2)$ and $(\}\{)$ can be the last rule in the proof for generating λ^-. In other words λ^- is a sub-message of a message in κ that can be decomposed, namely λ^- is in $\partial(\kappa)$. Therefore, $m \in \partial(\kappa)$ because the cryptogram can be disclosed. Hence, the $(\in)$ rule can be applied to $\partial(\kappa)$ and this completes the proof.

□

Finally, decidability of $\bowtie$ easily follows from computability of ∂ and decidability of $\rhd$, since Proposition 9.3.5 below holds.

Proposition 9.3.5 *If $\kappa \in \wp_{fin}(M)$ then $\partial(\kappa) \rhd m$ is decidable.*

PROOF. Only message of increasing size can be derived with rules $(,)$ and $(\{\})$. The number of messages having size less than or equal to m is finite, hence there is an effective procedure that can generate them all and check for equality with m. □

9.4 A Process Calculus for Security

This section introduces a calculus to describe security protocols. The calculus is a name-passing process calculus in the style of the π-calculus [130] with cryptographic primitives and mechanisms for sharing of keys and multiple sessions.

As already claimed, several generalization of the π-calculus have been proposed for studying many aspects of global computing (see Section 3.1). Some of them also consider cryptographic protocols. For instance, [2] extended π-calculus with cryptographic primitives thus providing a calculus for security protocols. Another extension in the same line is given in [22]

The calculus of *Interaction Patterns* (IP) of [29] extends π-calculus with mechanisms to describe coordination of interacting components inside open systems, i.e. systems whose structure is only partially specified. Interacting components are represented as instances of some predefined templates that may dynamically join a context of running components. Such idea is well suited to describe the intended execution model of cryptographic protocols where each "regular" principal behaves as prescribed by a template and many instances of the same template may be added to the execution context. However, the possibility of intruders that do not necessarily act in any prefixed way must be taken into account.

We introduce a calculus of *cryptographic interaction patterns* (cIP for short). The cIP calculus extends IP calculus in the same style of spi-calculus extension

to π-calculus. In fact, cryptographic primitives are added in order to deal with symmetric and asymmetric key protocols.

9.4.1 cIP syntax

We give the syntax and the preliminary definitions of the cIP calculus.

Definition 9.4.1 (Behavioural expressions) Behavioural expressions *are derived from the followings productions:*

$$\begin{array}{rcl} E, F & ::= & \mathbf{0} \mid \alpha.E \mid E \parallel E \mid E + E \\ \alpha, \beta & ::= & in(d) \mid out(d) \end{array}$$

where d is derived from the production for messages extended with productions for variables $M ::= x \mid ?x \mid [M]$.

Let X be the set of variables. A binding occurrence *of a variable x, is an occurrence of $?x$ in an input action $in(d)$.*

We assume that, for any variable x,

- *actions $out(d)$ do not contain any occurrence of $?x$ or $[M]$ (in d);*
- *actions $in(d)$ have at most one occurrence of $?x$ (in d).*

A datum d is a message containing variables or *syntactic tests*, namely, the complementary output action that matches $in([m])$ is $out(m)$. Hereafter, we let m to range over (ground) messages and let d to range over data.

Let us consider a datum of the form d_1, d_2. We say that variable occurrences of d_1 *precede* occurrences of variables in d_2. Moreover, given a behavioural expression of the form $in(d_1, d_2).E$, we say that the occurrences of variables in d_2 and in E are in the *scope* of a binding occurrences in d_1.

It is worth to remark that names and variables are syntactically distinguished. Names should be thought of as being constant terms whereas variables are placeholders and are amenable to be substituted with terms or opportunely renamed. In this respect, the following definition introduces free and bound variables.

Definition 9.4.2 (Bound and free variables) *An occurrence of a variable x is* bound *in E if it is in the scope of an input action $in(d)$ containing a binding occurrence of x. We denote with* $\mathrm{bn}(E)$ *the set of variables having bound occurrences in E. Occurrences of a variable x in E that are not bound are said* free occurrences *and* $\mathrm{fn}(E)$ *denotes the set of variables having free occurrences in E. A closed behavioural expression is an expression E such that* $\mathrm{fn}(E) = \emptyset$.

We consider equivalent behavioural expressions that differ only for bound variables. Indeed, as usual, we can always rename bound variables. Renaming of bound variables also permits us to assume that, given a behavioural expression

E, $\mathrm{bn}(E) \cap \mathrm{fn}(E) = \emptyset$. Moreover, we can assume that for any behavioural expression of the form $E_1 \parallel E_2$ or $E_1 + E_2$, $\mathrm{bn}(E_1) \cap \mathrm{bn}(E_2) = \emptyset$. Hereafter, we consider only behavioural expressions that satisfy the syntactic constraints above.

We can now formally define a principal, namely a participant of a protocol.

Definition 9.4.3 (Principal) *A* principal *is a triple written as* $A \triangleq (\tilde{X})[E]$ *where*

- $A \in N_p$,
- $\tilde{X}$ *is a tuple of variables (pairwise distinct), called* open variables *and*
- E *is a behavioural expression such that* $\mathrm{fn}(E) \subseteq \tilde{X}$.

We denote with $\mathrm{ov}(A)$ *the set of open variable* $\tilde{X}$ *of principal* A*, while set* $\mathrm{bn}(A) = \mathrm{bn}(E) \cup \tilde{X}$ *is the set of bound variable of* A.

We remark that principal names N_p are not used in behavioural expressions, in recursive call; they simply are names for finite patterns of communications and will also permit us to predicate on the identity of protocol participants. As for behavioural expressions, we assume that principal expressions are equivalent up to renaming of their bound variables. Notice that, by definition, $\mathrm{fn}(A) = \emptyset$ for each principal A.

Example 9.4.1 *Principals for Needham-Schroeder protocol are expressed in cIP as follows:*

$$\begin{array}{lcl} A & \triangleq & (y)[out(\{na, A\}_{y^+}).\ \ in(\{na, ?u\}_{A^-}).\ \ out(\{u\}_{y^+})] \\ B & \triangleq & ()[in(\{?x, ?z\}_{B^-}).\ \ out(\{x, nb\}_{z^+}).\ \ in(\{nb\}_{B^-})] \end{array} \tag{9.8}$$

where y *is the open variable denoting the responder.*

Definition of A *corresponds to the specification of the initiator given in Example 8.2.1; namely,* A *uses nonce* na *for authenticate itself to the responder represented by the open variable* y*.* A *first sends the pair* na, A*, encrypted under the public key of the responder* y^+*, then waits for a cryptogram obtained by encrypting with the public key of* A *the nonce* na *and another nonce received in the variable* u*.*

Actions of principals 9.8 that are in the same column synchronise once open variable y *of* A *is instantiated with* B*.*

Notice that the cryptogram in the in action of A *uses* A^-*. As will be explained later, this corresponds to require that the matching cryptogram is encrypted under* A^+*; in other words, the in action uses* A^- *to decrypt the received cryptogram. Similarly,* B *formalises the responder of Example 8.2.1 that performs the complementary actions of* A*.*

Principals interact by joining a context populated by other partners. Many instances of the same principal may non-deterministically access the execution environment. This amounts to say that more sessions of the protocol can be contemporary running. Moreover, new principal instances can dynamically join an execution environment.

Definition 9.4.4 (Principal instances) *Given a principal* $A \triangleq (\tilde{X})[E]$ *and a natural number* i*, an* instance of A *is the indexed principal* $A_i \triangleq (\tilde{X_i})[E_i]$*, which is obtained by indexing all variables and local names of* A*.*

Principal instances run in a *context*:

Definition 9.4.5 (Contexts) *A* context $\mathcal{C}$ *is a (possibly empty) set of principal instances. We let* $\mathrm{bn}(\mathcal{C})$ *and* $\mathrm{ov}(\mathcal{C})$ *respectively be the union of* bound *and* open variables *of principal instances in* $\mathcal{C}$*.*

A context can be dynamically extended introducing new instances of principals. This is modeled by means of a *join* operation, that takes care of connecting open variables of principal instances.

Definition 9.4.6 (Join) *Let* $A_n \triangleq (\tilde{X}_n)[E_n]$ *(*$n > 0$*) be a principal instance and let* $\mathcal{C}$ *be a context of cardinality* $n-1$*. Function* $\gamma : \mathrm{ov}(\mathcal{C}) \cup \tilde{X}_n \multimap (N_p \cup K)$ *is a partial mapping such that, for all* $x \in dom(\gamma)$*, either* $\gamma(x) \in K$*, if* x *is a variable for a symmetric key, or* $\gamma(x) \in N_p$*. The* join *operation is defined as:*

$$join(A_n, \gamma, \mathcal{C}) = (\tilde{X}_n - dom(\gamma))[E_n\gamma] \cup \bigcup_{(\tilde{Y})[F] \in \mathcal{C}} (\tilde{Y} - dom(\gamma))[F\gamma]$$

The join operation defines how a principal instance can enter a (running) context by connecting open variables for asymmetric keys to principal names and open variables for symmetric keys to keys K so that they are appropriately shared. Connected variables are no longer open. A cIP context is an environment where computation takes place and new processes may be dynamically added.

9.4.2 cIP semantics

The semantics of our calculus is given by means of two transition systems, $\rightarrow$ and $\mapsto$ respectively. Labelled transition system $\rightarrow$, defined up to structural congruence[1], models the "stand alone" behaviour of principals. Reduction relation $\mapsto$ models both communications taking place inside contexts and possible evolutions of a context due to the joining of a new principal instances.

Definition 9.4.7 (Behavioural expression semantics) *Let* E *be a behavioural expression. Labelled transition system* $\rightarrow$ *is defined by induction on the structure of* E *by means of the following inference rules:*

$$\frac{}{\alpha.E \xrightarrow{\alpha} E}\ (pre) \qquad \frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}\ (+)$$

$$\frac{E \xrightarrow{\alpha} E'}{E \parallel F \xrightarrow{\alpha} E' \parallel F}\ (\parallel) \quad \mathrm{bn}(\alpha) \cap \mathrm{fn}(F) = \emptyset$$

[1]Structural congruence, $\equiv$, is obtained by extending α-conversion with $\parallel$ and $+$ as monoidal operators, whose neutral element is $\mathbf{0}$.

The rules above are quite similar to the corresponding rules of π-calculus (Section 3.1).

Definition 9.4.8 (Valid substitutions) *Given $m \in M$ and a behavioural expression E, a substitution σ is* valid for m in E *if, and only if, σ maps all variable that occur in E as encryption keys into K.*

Definition 9.4.8 states that variables denoting keys are substituted with keys. Hereafter, we assume that all substitutions are valid.

Observation 9.4.1 *The assumption on valid substitution can be adopted if we assume that the underlying cryptographic system provide enough information in data exchanged in communications in order to check whether a message has or not an expected form. This hypothesis can be relaxed at the cost of adding a major complexity in definitions and proofs of the results. Here, we prefer to maintain the presentation as simple as possible.*

Before entering into the details of the contexts semantics, we define the pattern matching mechanism. Message matching is required because an input action $in(d)$ can be fired only if the message acquired matches the pattern described by d.

Definition 9.4.9 (Matching) *Let m and n be two messages. We say that m and n* match *($m \sim n$) if, and only if,*

- *if $m, n \in N \cup K$ then $m = n$;*
- *if $m = [m']$, then n is syntactically equal to m;*
- *if $m = p, q$ then $n = p', q'$ and $p \sim p'$ and $q \sim q'$;*
- *if $m = \{m'\}_\lambda$ and $n = \{n'\}_{\lambda^-}$ then $m' \sim n'$.*

A datum d matches *a message m if, and only if, there exists a substitution σ over the variables occurring in d such that $d\sigma \sim m$.*

The first two clauses of Definition 9.4.9 are trivial. The third clause deals with cryptograms. The intuition is that two $m = \{m'\}_\lambda$ and $n = \{n'\}_{\lambda^-}$ cryptograms match if m' and n' do match and the key of m is the inverse of the key of n. In the case of symmetric keys this simply reduces to require that the encryption keys are equal. For instance, $\{na\}_k$ matches itself. As intuitively described in Example 9.4.1, in the case of asymmetric keys Definition 9.4.9 states that m matches n when the key of m is the public (private) key of a principal and the key of n is the private (public) key of the same principal. For instance, $\{A\}_{B^-}$ matches $\{A\}_{B^+}$.

Definition 9.4.9 becomes more clear if we consider that, in cIP, encryption and decryption mechanisms are embodied into the *in* and *out* actions. For instance, the

$$\frac{E_i \xrightarrow{in(d)} E_i' \qquad \partial(\kappa) \rhd m : \ \exists\sigma \text{ ground s.t. } d\sigma \sim m}{\langle(\tilde{X}_i)[E_i] \cup \mathcal{C}, \chi, \kappa\rangle \ \mapsto \ \langle(\tilde{X}_i)[E_i'\sigma] \cup \mathcal{C}, \chi\sigma, \kappa\rangle}\ (in)$$

$$\frac{E_i \xrightarrow{out(m)} E_i'}{\langle(\tilde{X}_i)[E_i] \cup \mathcal{C}, \chi, \kappa\rangle \ \mapsto \ \langle(\tilde{X}_i)[E_i'] \cup \mathcal{C}, \chi, \kappa \cup m\rangle}\ (out)$$

$$\frac{\mathcal{C}' = join(A_i, \gamma, \mathcal{C}) \qquad A \triangleq (\tilde{X})[E] \qquad i \text{ new}}{\langle\mathcal{C}, \chi, \kappa\rangle \ \mapsto \ \langle\mathcal{C}', \chi\gamma, \kappa \cup \{A_i\}\rangle}\ (join)$$

Table 9.1: Context reduction semantics

in action of principal A in Example 9.4.1 waits for a datum encrypted with the public key of A, hence, the decryption key used is A^-, the private key of A.

Binding occurrences in data used inside input actions can be thought of as patterns that must match data of output actions. If the matching holds then the synchronization can take place and the binding variables are substituted with corresponding values in output data.

Semantics of contexts aims at formalizing the behaviour of the Dolev-Yao intruder. The intruder keeps track of principal activities, namely, the messages which have been exchanged among principals. However, contexts also maintain information about connections of open variables of principal instances. This is necessary for verification purposes as will be clearer later. Relation $\mapsto$ (defined below) expresses reductions of *configurations* $\langle\mathcal{C}, \chi, \kappa\rangle$ where $\mathcal{C}$ is a context, χ is a variable binding that keeps track of the associations of the variables due to communications and join executions, and κ contains the instance names that joined the context and the data sent along the public channel, i.e. κ represents the intruder knowledge.

Definition 9.4.10 (Context reduction) *The* context reduction *relation is the smallest binary relation between contexts induced by the inference rules in Table 9.1.*

If a context contains a principal instance waiting for a message and the knowledge of the environment can generate a message that matches d via a ground substitution σ, then the system may evolve by applying σ to the continuation of the principal instance and storing bindings determined by σ (rule (in)).
Rule (out) simply states that a message sent by a principal instance is recorded in κ. Notice that, the hypothesis of using only closed principals guarantees that only ground data, i.e. messages, are sent by principals.
Rule $(join)$ adds a new instance of a principal to the context. Notice that a "bad" connector γ may yield a "wrong" context where keys are not shared in the intended way. Moreover, note that the intruder gets aware of the entering of new instances: A_i is added to κ. Rule $(join)$ provides a mechanism to express the dynamic composition of components to a running open context.

Observation 9.4.2 *Without loss of generality, we assume that the intruder knows* all *the public keys of regular principals. Under this assumption the rule* (*join*) *becomes*

$$\frac{\mathcal{C}' = join(A_i, \gamma, \mathcal{C}) \qquad A \stackrel{\triangle}{=} (\tilde{X})[E] \qquad i \text{ new}}{\langle \mathcal{C}, \chi, \kappa \rangle \;\mapsto\; \langle \mathcal{C}', \chi\gamma, \kappa \cup \{A_i, A_i^+\} \rangle} \;.$$

However, to have a more compact representation of κ *we implicitly assume that* A_i^+ *is in* κ *whenever* κ *contains* A_i*, avoiding to explicitly write* A_i^+*.*

9.4.3 Discussion

One of the most important characteristic of cIP calculus is the smooth distinction between the "static" aspects of protocols and their dynamic behaviour that also suffers the distinction between instances of some rôle. This will be exploited in Section 9.5 where a logic for reasoning about and verifying properties on protocols will be introduced. If we focus our attention on the informal description of security issues outlined in Section 3.5 then we can say that cIP matches many of the requirements usually stated on security protocols. Indeed, a natural correspondence exists between informal specifications and cIP formalizations. In order to illustrate this relation, let us reconsider the Needham-Schroeder protocol. We recall that the informal specification together with its correspondent formalization:

$$\begin{array}{lll} (1) \quad A \to B : \{na, A\}_{B^+} & A & \stackrel{\triangle}{=} \quad (y)[out(\{na, A\}_{y^+}).in(\{na, ?u\}_{A^-}).out(\{u\}_{y^+})] \\ (2) \quad B \to A : \{na, nb\}_{A^+} & B & \stackrel{\triangle}{=} \quad ()[in(\{?x, ?z\}_{B^-}).out(\{x, nb\}_{z^+}).in(\{nb\}_{B^-})]. \\ (3) \quad A \to B : \{nb\}_{B^+}. & & \end{array}$$

Each principal corresponds to a rôle in the protocol and its behaviour at each step depends on whether the principal is sending or receiving a message. In general, names appearing in a message are considered local to the first principal who sent them. Indeed, the receiver "reads" such names in its own local variables. For instance, principal B acts as responder and reads nonce na originated by A in its variable x bound by its first input action. Then, the second action replies the pair x, nb to A where x contains the nonce na and nb is a nonce generated by B. In general, the protocol designer must specify open variables of principal. Even though it seems that there is no algorithmic way to define principals from their informal specifications, we can try to give some rule of thumbs:

- initiator usually need an open variable that responder should join;
- if the identity of the partner is acquired in a communication, probably no open variable is required from that partner;
- an open variable might be necessary when a principal must interact with a server.

It is easy to observe, that the above listed rules fit well with the specification of the initiator and responder of the Needham-Schroeder protocol.

We end this section by briefly summarizing the main differences between cIP and π-calculus.

Implicit Restriction. All names occurring in a behavioural expression must be considered as restricted and, as usual, two behavioural expressions are equivalent if one is obtained by alpha-renaming the private names of the other.

Name Usage. cIP models cryptographic keys and principal identities as names, differently from π-calculus where names are abstractions for communication channels. Communication between cIP principals is anonymous, namely principals do not refer any channel name and synchronizations take place along a unique public channel where everyone may send or receive. Name extrusion still remains an important feature of our calculus because principals exchange keys in order to establish a "secure channel" for communicating their secrets along a public channel.

Matching Communications. The cIP calculus exploits matching in its communication primitives: When two processes synchronize, the term appearing in the input process must *match* the sent message. Hence, synchronization encapsulates encryption and decryption mechanisms via matching.

Absence of Iteration or Recursion. The cIP calculus has been designed to specify finite deterministic protocols. Hence, replication and recursion have not been considered as basic primitives of the calculus. Indeed the calculus can handle a wide class of cryptographic protocols. Moreover, new instances of principal may be non-deterministically added to a context. Semantically, the join mechanisms corresponds to replication.

Variables and Names. Usually, π-calculus do not make any distinction between variables and names. Following [2], we separate the two concepts for two reasons. The first is that not only names (i.e. keys) may be communicated but also complex terms such as tuples or cryptograms. Second, we will assume that each principal instance that takes part in a computation uses private variables that may be instantiated to names shared with other participants.

9.5 Formalizing Security Properties

This section presents a logical language to specify security properties of protocols. The logic allows one to express properties concerning values that variables are supposed to assume, messages that must or must not belong to the intruder knowledge and relations among the values shared by different principal instances. In this logic,

integrity corresponds to the possibility of fixing some value, generalizing the approach introduced in [2]. Secrecy is handled by exploiting intruder knowledge and the values it may or may not contain. Many authors reduce authentication to causality relations over the structure of the events observed in the state space of the computation.

The design of our logic is guided by the features of the cIP calculus; in particular, cIP offers the possibility of uniformly extending a context with new instances of principals of the protocol. What here is meant by "uniformly" is the fact that variables occurring in principal expressions are not renamed when a new instance joins the context but they are simply labelled with a unique index. This linguistic mechanism allows us to determine which are the instances that originated the names used through the execution of the protocol as well as to distinguish between different participants that play the same rôle. Informally, security properties express relations between instances and their variables; therefore, we design a logic which can explicitly talk about them. In the previous discussion it emerges that rôles can be viewed as "templates" or "types" that specify both particular behaviours of principals and uniformly describe variables that occur in instances obtained by instantiating rôles.

Definition 9.5.1 ($\mathcal{PL}$ – Syntax) *A formula of the logic $\mathcal{PL}$ (*protocol logic*) is defined as follows:*

$$\begin{array}{rcl} \phi, \psi & ::= & \delta \in \mathfrak{K} \mid \alpha = \beta \mid x@\alpha = \delta \\ & \mid & \forall \alpha : A.\phi \mid \neg\phi \mid \phi \wedge \psi \\ \\ \delta & ::= & d \mid \alpha \mid x@\alpha \mid \mathbf{I} \end{array} \tag{9.9}$$

where d is datum that does not contain any binding occurrence. Formulae of productions 9.9 are called positive atoms *while their negations are called* negative atoms*.*

Operators $\neg$, $\wedge$ and $\vee$ are the usual boolean operators. Derived relations $\neq$ and $\notin$, logical connectors $\rightarrow$ and $\vee$, or existential quantifier $\exists$ are defined as usual and will be used as syntactic sugar. The symbol $\mathfrak{K}$ is used to represent the knowledge that the intruder acquires during a protocol computation.

Notation 9.5.1 *We use $\delta \in \mathfrak{K}$ in Definition 9.5.1 for expressing that δ is known to the intruder. We believe that, despite of its ambiguity ($\in$ is only a syntactic symbol), this notation does not create any confusion, still preserving an intuitive understanding for such kind of formulae.*

$\mathcal{PL}$ syntax introduces a new class of variables which are the *instance variables* α, β, etc., that are subject to equality check and quantification. Instance variables range over (indexed) instances of rôles and are "typed" by principal names. For instance, proposition $\forall \alpha : A.\phi$ reads as "for all instances of A, ϕ holds". Instances

univocally determine participants to a protocol session and, assuming (as we did) that principals are defined such that all binding occurrences differ each other[2], expression $x@\alpha$ selects the content of variable x of instance α.

Remark 9.5.1 *Only instance variables are considered as variable symbols in the logic. Namely, x's in Definition 9.5.1 should be thought of as symbols for constants. Hence, $\forall\alpha : A.x@\alpha \in \mathfrak{K}$ is a closed formula, while $\forall\alpha : A.\alpha = \beta$ is open because β is not in the scope of a quantifier.*

Among the possible value that can be expressed in the formulae there is the special constant **I** that denotes the intruder's identity. This permits expressing propositions where the identity of the partner is not necessarily a protocol rôle. For instance, we can require that each initiator of the Needham-Schroeder public key protocol interact with a B, unless the initiator starts communicating with **I**. The following example should clarify this point.

Example 9.5.1 *A property that the protocol should satisfy is the secrecy of the nonces na and nb. It is evident that the latter is secret unless it is produced by the intruder. Therefore, the formula*

$$\forall\alpha : A.u@\alpha \in \mathfrak{K} \rightarrow y@\alpha = \mathbf{I}$$

holds if A starts interacting with the intruder.

Example 9.5.2 *The intended use of the Needham-Schroeder protocol (see Example 8.3.1, page 143) is the* weak agreement *property that has been pointed out in [115] and can be informally summarized as:*

> *If B completes a protocol session and thinks he has been talked to A then A started the protocol session thinking to have been talked to B.*

We may formalize this statement with a $\mathcal{PL}$-formula ϕ_{NS}:

$$\phi_{NS} \equiv \forall\beta : B.(nb@\beta \in \mathfrak{K} \rightarrow z@\beta = \mathbf{I}) \vee \exists\alpha : A.(z@\beta = \alpha \rightarrow y@\alpha = \beta)$$

which reads: For each B_i, instance of B, either the intruder obtains nb_i because he starts talking to B_i or there exists A_j, instance of A, such that, if the partner of B_i had communicated with A_j then A_j had communicated with B_i (in the same protocol session).

Before proceeding in defining models for $\mathcal{PL}$-propositions, we must define how values are substituted for binding variables in expressions δ.

[2]This is not a restrictive hypothesis because we can always α-rename binding occurrences with new names.

Definition 9.5.2 (Binding variable substitutions) *An expression δ is* ground *if it does not contains instance variables. Let χ be a mapping defined on indexed variables occurring in a ground expression δ, we define the application of substitution χ to δ by induction on the structure of δ:*

$$\delta\chi = \begin{cases} d\chi, & \text{if } \delta = d \\ A_i, & \text{if } \delta = A_i \\ x_i\chi, & \text{if } \delta = x@A_i. \end{cases}$$

Definition of $\delta\chi$ states that the substitution χ is applied to non-instance variables of that occur in δ. We remark that such symbols are considered as symbols for constants in $\mathcal{PL}$, indeed there are two levels of substitutions:

- a first level will be considered in Definition 9.5.3 and instantiates instance variables when quantifiers are eliminated from the formula;
- the second level of substitutions is required to determine the values of local variables of principals as defined in Definition 9.5.2.

Formulae are verified with respect to a given (terminating) state of the computation of a context, and to the instances that actually have participated in the computation. We adopt the notation $\kappa \models_\chi \phi$ to indicate that the set κ, under the variable assignment specified by χ, is a model of the formula ϕ. We define models for *closed* $\mathcal{PL}$-formulae.

Definition 9.5.3 (Models of $\mathcal{PL}$) *Let χ be a ground substitution from variables X. A* model for a $\mathcal{PL}$ closed formula ϕ *is a pair $\langle\kappa, \chi\rangle$ such that $\kappa \models_\chi \phi$ can be proved by the following rules:*

$$\frac{i = j}{\kappa \models_\chi A_i = A_j}\,(=_1) \qquad \frac{x_i\chi = \delta\chi}{\kappa \models_\chi x@A_i = \delta}\,(=_2) \qquad \frac{\partial(\kappa) \rhd \delta\chi}{\kappa \models_\chi \delta \in \mathfrak{K}}\,(\in)$$

$$\frac{\kappa \models_\chi \phi \quad \kappa \models_\chi \psi}{\kappa \models_\chi \phi \wedge \psi}\,(\wedge) \qquad \frac{\kappa \not\models_\chi \phi}{\kappa \models_\chi \neg\phi}\,(\neg)$$

$$\frac{\kappa \models_\chi \phi[^{A_j}/_\alpha] \;\; \textit{for all } A_j \in \partial(\kappa)}{\kappa \models_\chi \forall\alpha : A.\phi}\,(\forall).$$

Rule $(=_1)$ says that $\langle\kappa, \chi\rangle$ is a model of equality $A_i = A_j$ whether the instances are exactly the same instance. Rule $(=_2)$ says that $\langle\kappa, \chi\rangle$ is a model of $x@A_i = \delta$ whether the value associate by χ to the x variable of instance A_i, i.e. $x_i\chi$ equals the valued $\delta\chi$. Rule $(\in)$ establishes that $\kappa \models_\chi \delta \in \mathfrak{K}$ when $\delta\chi$ can be constructed from the decomposition set of κ. Rules $(\wedge)$ and $(\neg)$ are trivial. In $\forall\alpha : A.\phi$ the universal quantifier ranges over the finite set of instances of rôle A. Quantifiers are resolved by relating variables to actual instances. In order to prove that $\langle\kappa, \chi\rangle$ is a model for

a formula $\forall\alpha : A.\phi$, it is necessary to show that $\langle\kappa, \chi\rangle$ is a model for any formula obtained by substituting A_j for α in ϕ, where A_j is any instance of A deducible from κ.

Definition 9.5.3 is based on a *close world assumption*: κ is a model for a formula ϕ with respect to χ if and only if there exists a proof for $\kappa \models_\chi \phi$. This justifies the symbol $\not\models$ in rule $(\neg)$, which reads as "κ is a model for $\neg\phi$ with respect to χ if κ *is not* a model for ϕ" with respect to χ, i.e. if does not exist a proof for $\kappa \models_\chi \phi$. Relations $\rhd$ and $=$ are decidable. It follows that $\models$ and, hence, $\not\models$ are decidable too.

Notice that if χ and χ' differ only on variables not appearing in ϕ, then $\kappa \models_\chi \phi \Leftrightarrow \kappa \models_{\chi'} \phi$. Hence, we can only consider finite assignments over the variables of ϕ.

Chapter 10

Toward Algorithmic Verification

Abstract

In this chapter, we provide a symbolic semantics for cIP. Symbolic semantics is more suitable for automatic verification since it allows one to generate a transition system which is finite (up to the number of possible join transitions). Hence the symbolic transition systems can be used to model check $\mathcal{PL}$ formulas expressing (security) properties cIP principals.

This chapter is organized as follows: First, we discuss which are the symbolic messages that can appear in a communication and how they are constrained by the evolution of the computation. Then, we define how a symbolic intruder can be algorithmically constructed. We show how security properties can be verified with respect to symbolic traces. Finally, we prove a correspondence theorem between symbolic semantics and the "concrete" semantics of Section 9.4.

Contents

10.1 Symbolic Intruder

By interacting with other principals, the intruder can either extend its knowledge by receiving a message or it can send a message deducible from its current knowledge. In the latter case, the intruder sends a message m that is required to match a datum d of an input action of a principal, via a ground substitution. Thanks to the matching mechanism, d constrains the choices for m, reducing the set of the possible messages the intruder can deduce from its knowledge to send them to regular principals. Unfortunately, if d contains binding occurrences of variables, the intruder could in principle send an infinite number of messages even if κ is finite. For instance, from $\kappa = \{k\}$, the intruder can generate $\{k\}_k, \{\{k\}_k\}_k, ...$ all matching $\{?x\}_k$.

Notice how the phenomenon detailed above is similar to the infinite branching induced by the input actions in the early semantics of π-calculus discussed in Observation 3.1.1 (page 44). This motivates symbolic analysis: No message is chosen, but rather a finite representation of κ is sent. This is a sort of *lazy evaluation*, even if no choice is done, all possibilities are preserved by associating the current knowledge κ to bound variables of input actions.

Definition 10.1.1 (Symbolic messages) *The set of* symbolic messages $\underline{M}$ *is generated by the following productions:*

$$\underline{M} \;::=\; N \;\mid\; K \;\mid\; \underline{M},\underline{M} \;\mid\; \{\underline{M}\}_{\underline{M}} \;\mid\; x(\kappa) \;\mid\; \hat{x}(\kappa),$$

where $\kappa \in \wp_{fin}(\underline{M})$ *and* $x \in X$*. Basically, the above grammar extends (concrete) messages with the productions for* symbolic variables*. Observe that, by Definition 9.2.1,* $M \subseteq \underline{M}$*, hence we still let* m *to range over* $\underline{M}$*.*

Symbolic *marked* variables $\hat{x}(\kappa)$ are introduced to limit the set of values that can be assigned to $\hat{x}(\kappa)$. Suppose that two actions $in(x(\kappa))$ and $out(x(\kappa))$ synchronize and that $\kappa = \{no, A, A^+\}$. While the (concrete) choice of A^+ for x is correct, since $A^+ \sim A^+$ (see Definition 9.4.9, page 9.4.9), the message $\{no\}_{A^+}$ should not be considered as one of the possible matching messages (even though it can be derived from κ) because $\{no\}_{A^+} \not\sim \{no\}_{A^+}$ (but rather $\{no\}_{A^+} \sim \{no\}_{A^-}$). This is resolved by marking $x(\kappa)$ so that substitution will disregard "undesired" values (see Definition 10.1.3).

Derivation from a set of symbolic messages is defined similarly to Definition 9.3.4.

Definition 10.1.2 (Symbolic message construction) *A symbolic message* $m \in \underline{M}$ *can be* constructed *from set* $\kappa \subseteq \underline{M}$*, if* $\kappa \trianglerighteq m$ *can be proved by the following rules:*

$$\frac{m \in \kappa}{\kappa \trianglerighteq m} \qquad \frac{\kappa \trianglerighteq m \quad \kappa \trianglerighteq n}{\kappa \trianglerighteq m, n} \qquad \frac{\kappa \trianglerighteq m \quad \kappa \trianglerighteq \lambda}{\kappa \trianglerighteq \{m\}_\lambda}$$

$$\frac{\kappa \trianglerighteq m \;\; \textit{for all } m \in \kappa'}{\kappa \trianglerighteq x(\kappa')} \qquad \frac{\kappa \trianglerighteq m \;\; \textit{for all } m \in \kappa'}{\kappa \trianglerighteq \hat{x}(\kappa')}$$

The last two rules allow the intruder to generate symbolic variables $x(\kappa')$ and $\hat{x}(\kappa')$ from κ and to recognize when the set of messages derivable from κ' is contained in the set of messages derivable from κ.

Example 10.1.1 *Let us consider the sets $\kappa' = \{\{no_1\}_k\}$ and $\kappa = \{no_1, k\}$. Clearly, the set of messages derivable from κ' is included in the set of messages derivable from κ, but the inverse inclusion does not hold because $k \notin \kappa'$.*

To guarantee that all values represented by a symbolic message can be communicated in a concrete trace, it is necessary to keep symbolic variables consistent with their use in the protocol, so that, for example, a symbolic variables used as a key must be instantiated only to keys. In particular, since different choices for a key may lead to different concrete evolutions, we require that symbolic values do not occur as keys. We assume, therefore, that messages contains enough information for self-describing their "type".

Validity of the formulae expressing security properties will be checked in terminal states according to the notion of symbolic model introduced later. If a formula depends on a symbolic variable, e.g. $x(\kappa)$, the formula itself will help in deducing the correct message derivable from κ, if any, that can be associated to $x(\kappa)$ and which makes the formula true. It will be proved that no information is lost in this process: A terminal state of a symbolic evolution that satisfies ϕ exists if and only if a terminal (concrete) state satisfying ϕ exists.

Definition 10.1.3 (Symbolic substitution) *Let $\underline{X}$ be the set of symbolic variables (marked or not); a partial function $\sigma : X \cup \underline{X} \to \underline{M}$ is a* symbolic substitution *if,*

- *for each $x(\kappa) \in \underline{X}$, $\kappa \trianglerighteq (x(\kappa))\sigma$ and*
- *for each $\hat{x}(\kappa) \in \underline{X}$, any sub-term of $(\hat{x}(\kappa))\sigma$ is not a cryptogram constructed with asymmetric keys, and $\kappa \trianglerighteq (\hat{x}(\kappa))\sigma$.*

Actions $out(d)$ and $in(d')$ can synchronize only if there exists a suitable (symbolic) substitution such that $d\sigma, d'\sigma \in \underline{M}$ match. Symbolic communications resume all the communications that can be obtained by instantiating a symbolic variable with one of the messages (derivable from the set κ). To enforce this property, some care is necessary in defining when two symbolic messages match, so that all symbolic traces correspond exactly to possible corresponding concrete traces. Moreover, the evolution of a trace can further constrain a symbolic variable according to its usage. Let us consider $in(?x).in(\{?y\}_x)$ and assume that the action $in(?x)$ binds $?x$ to $x(\{no, k\})$, where no is a nonce and k a key. The action $in(\{?y\}_x)$ constrains $x(\{no, k\})$ to k, the only value that can be derived from κ and that is a key. Evolutions of a context proceed according to new notion of *symbolic matching*.

Definition 10.1.4 (Symbolic matching) *Given $m, n \in \underline{M}$ we say that m and n* symbolically match *(and write $m \simeq n$) if, and only if, one of the following alternatives applies:*

1. $m = n = \hat{x}(\kappa)$
2. $m = p, q \ \wedge \ n = p', q' \ \wedge \ p \simeq p' \wedge \ q \simeq q'$
3. $m = \{m'\}_\lambda \ \wedge \ n = \{n'\}_{\lambda^-} \ \wedge \ m' \simeq n'$
4. $m = n \ \wedge \ m, n \in N \cup K$

Case (1) deals with matching symbolic variable $\hat{x}(\kappa)$ with itself and it requires the particular restriction to $x(\kappa)$ previously discussed. Case (2) states that two pairs match if their respective components symbolically match. Case (3) deals with symbolic cryptograms. Due to our choice of embedding decryption mechanisms into communication, a cryptogram with key λ can match only a cryptogram that exhibits a complementary key λ^-, and whose messages match one another. Case (4) states that symbolic matching reduces syntactic equality when names or keys are considered.

Two different symbolic variables $x(\kappa)$ and $y(\kappa')$ match provided that they represent the same choices of concrete messages. In other words, they should have been equated to the same symbolic variable by an appropriate substitution σ, i.e. $x(\kappa)\sigma \simeq y(\kappa')\sigma$. Matching of a symbolic variable $x(\kappa)$ and a message m depends on the existence of an appropriate substitution such that $x(\kappa)\sigma \simeq m\sigma$. These cases are not dealt with in Definition 10.1.4 because the symbolic semantics of cIP (Definition 10.1.6) both messages and data of complementary input/output actions are subject to substitutions that avoid these case to appear.

Symbolic semantics requires some care when messages are sent by principals. A principal may send a message containing symbolic variables (acquired in previous input actions). For instance, assume that $in(?x).out(\{no\}_x).E$ is the behavioural expression of a principal, then, if the input action produces the substitution $[^{x(\{k\})}/_x]$, the continuation of the principal is $out(\{no\}_{x(\{k\})})$. Although substitution $[^{k}/_{x(\{k\})}]$ is a "valid" assignment for $x(\{k\})$, symbolic substitutions exist that do not correspond to any concrete trace: Indeed $[^{k,k}/_{x(\{k\})}]$ would add $\{no\}_{k,k}$ to the intruder knowledge.

Definition 10.1.5 (Valid symbolic substitution) *Given a symbolic message m and a behavioural expression E, a symbolic substitution σ is* valid for m in E *if, and only if, all symbolic variable $x(\kappa)$ that occur in E or in m as encryption key are mapped by σ in $K \cap \kappa$.*

Observation 10.1.1 *Notice that given $m \in \underline{M}$, for each $x(\kappa)$ occurring in m, there is only a finite number of possibilities for mapping $x(\kappa)$ into a key belonging to κ because κ is finite.*

$$\frac{E_i \xrightarrow{out(m)} E_i'}{\langle (\tilde{X}_i)[E_i] \cup \mathcal{C}, \chi, \kappa \rangle \overset{i\ m}{\rightsquigarrow} \langle \{(\tilde{X}_i)[E_i']\} \cup \mathcal{C}, \chi, \kappa \cup m) \rangle} \ (\underline{out})$$

$$\frac{E_i \xrightarrow{in(d)} E_i' \qquad \partial(\kappa) \trianglerighteq m \qquad \sigma \text{ valid symb. substit. for } m \text{ in } E_i \text{ s.t. } m\sigma \simeq d\sigma}{\langle (\tilde{X}_i)[E_i] \cup \mathcal{C}, \chi, \kappa \rangle \overset{o\ m\sigma}{\rightsquigarrow} \langle \{(\tilde{X}_i)[E_i'\sigma]\} \cup \mathcal{C}\sigma, \chi\sigma, \kappa\sigma \rangle} \ (\underline{in})$$

$$\frac{\mathcal{C}' = join(A_i, \gamma, \mathcal{C}) \qquad A_i \triangleq (\tilde{X}_i)[E_i] \qquad i \text{ new}}{\langle \mathcal{C}, \chi, \kappa \rangle \overset{j\ A_i\ \gamma}{\rightsquigarrow} \langle \mathcal{C}', \chi\gamma, \kappa \cup A_i \rangle} \ (\underline{join})$$

Table 10.1: Context symbolic semantics

Given $x(\kappa) \in \underline{X}$, by Observation 10.1.1, we can define the finite set $\mathcal{B}(x(\kappa))$ containing all possible bindings $[^{\lambda}/_{x(\kappa)}]$, where λ is a key in κ. Similarly, for $m \in \underline{M}$, we can define $\mathcal{B}(m)$ as

$$\mathcal{B}(m) = \{\sigma_1 ... \sigma_n : \mathrm{n}(m) \cap \underline{X} = \{x_1(\kappa_1), .., x_n(\kappa_n)\} \ \wedge \ \sigma_i \in \mathcal{B}(x_i(\kappa_i)), i = 1, ..., n\},$$

namely, $\mathcal{B}(m)$ contains the substitutions of symbolic variables in m obtained by combining the substitutions of symbolic variables in m with the keys contained in the sets associated to the variables.

We can now introduce the symbolic semantics. States of the transition system (ranged over by Σ) are triples $\langle \mathcal{C}, \chi, \kappa \rangle$ where $\mathcal{C}$ is a context containing principals that may exchange symbolic messages; $\chi : X \cup \underline{X} \to \underline{M}$ is a symbolic substitution and $\kappa \in \wp_{\mathrm{fin}}(\underline{M})$ is the (finite) set of messages that the intruder knows. Despite the overloaded notation with respect to Definition 9.4.10, we remark that also symbolic messages must be considered in contexts, in bindings and in the intruder knowledge.

Definition 10.1.6 (Symbolic context semantics) *The* symbolic context semantics *relation is the smallest relation induced by the inference rules in Table 10.1.*

Rule ($\underline{out}$) says that each message sent by a principal is added to the intruder knowledge κ.

Notice that in rule ($\underline{in}$) the chosen message m is derived from κ and must symbolically match d via a substitution σ that constraints some variables. The message m is rendered by substituting all symbolic variables used as key in E_i, $x_i(\kappa')$, with a key derivable from κ'. The choice of σ, makes the ($\underline{in}$) rule a source of non-determinism. Observation 10.1.1 ensures that $\mathcal{B}(\_)$ is always a finite set, hence the possible evolutions remain a finite number. Essentially, σ represents constraints on the symbolic variable of its domain. These constraints are propagated to each principals of the context. Moreover, they are recorded in the bindings $\chi\sigma$ and update the knowledge.

We will define a function that takes κ and m' and gives back a message together with a substitution that satisfies the premise of ($\underline{in}$). This ensures that messages the intruder sends can pass the matching phase of the communications.

Observation 10.1.2 *Inference rules in Table 10.1 preserves a useful property of symbolic messages in κ, namely, a sort of "monotonicity" of the set of messages associated to symbolic variables. We say that κ* covers *κ' if*

$$\forall m' \in \kappa' : \kappa \trianglerighteq m'.$$

If $x(\kappa')$ is a symbolic variable and κ covers κ', then we say that κ covers $x(\kappa')$ and, similarly, κ covers m if either κ covers all the symbolic variable occurring in m or $\kappa \trianglerighteq m$ whenever $m \in M$. Notice that, if κ covers any message in κ, then also the target state has a knowledge that covers its symbolic variables. Indeed, by inspecting rules of the symbolic semantics, either a message is generated from κ or a message m is added to κ.

Since we will consider traces that start from states having concrete knowledge, hereafter we assume that any symbolic knowledge enjoys the cover property of Observation 10.1.2.

We remark that (*in*) gives rise to a finite number of possible transitions: The number of substitutions and messages that satisfy its premise is finite thanks to the symbolic framework. Namely, when binding occurrences of variables must be matched, substitutions map them in corresponding symbolic variables.

Finally, in rule (*join*) when a new principal joins a context, it is instantiated to a regular principal and, therefore, it still does not contain any symbolic message.

Remark 10.1.1 *Given any context, both the reduction and the labelled transition semantics generate infinite number of (possibly) terminating traces. Indeed, any context may be non-deterministically extended with new principals added by join transitions. This problem is typically addressed by* explicitly *considering a final number of principal instances [132, 101, 22]. We will show how our approach* implicitly *handles (parametric) finite multi-sessions. Indeed, multi-session attacks constitute an important class of attacks. For instance [125] provided a protocol that fails to satisfy a security property only if multi-session attacks are considered.*

Observation 10.1.3 *The search space induced by the LTS can be suitably reduced by employing a priority policy on rule application. More precisely, consider an intruder that, in each state grabs all the messages that a regular principal sends. When no more messages can be collected, the intruder non-deterministically either chooses a message to send to waiting principals or let a new principal enter the running context. This correspond to give maximum priority to (out) and the same lower priority to (in) and (join).*

However, such a priority policy does not spoil the completeness of the analysis. Intuitively, an intruder that "anticipates" its output operations has less knowledge for generating its messages, hence he cannot be more "powerful" than an intruder that collects all principals' outputs before sending messages.

$$\mu(d,\kappa) = \begin{cases} \{(x,\ [^{x(\kappa)}/_{x}])\}, & d = ?x \\ \{(d,\varepsilon)\}, & d \in N \cup K \ \wedge\ \partial(\kappa) \trianglerighteq d \\ \{(x(\kappa'),[^{\hat{x}(\kappa')}/_{x(\kappa')}])\}, & d = x(\kappa') \ \wedge\ \partial(\kappa) \trianglerighteq x(\kappa') \\ \left\{ (e', f', \sigma_e\sigma_f) : \begin{array}{l} (e',\sigma_e) \in \mu(e,\kappa) \wedge \\ (f',\sigma_f) \in \mu(f\sigma_e, \kappa\sigma_e) \wedge \\ \sigma_e\sigma_f \neq \bot \end{array} \right\}, & d = e, f \\ \left\{ (\{e'\}_\lambda, \sigma_e) : \begin{array}{l} (\kappa \trianglerighteq \lambda \wedge (e',\sigma_e) \in \mu(e,\kappa)) \ \vee \\ (\{e'\}_\lambda \in \kappa \wedge \sigma_e \in \nu(e',e) \neq \bot) \end{array} \right\}, & d = \{e\}_{\lambda^-} \\ \emptyset, & otherwise \end{cases}$$

Table 10.2: Definition of μ

10.2 Output Messages

The choice of a message m derivable from the current knowledge κ which can match an input datum d, is driven by d itself. This is done by means of a function μ, which given d and κ yields a set of pairs (m, σ) where m is a message and σ a substitution such that $\kappa \trianglerighteq m$ and $m\sigma \simeq d\sigma$.

Definition 10.2.1 (Intruder output messages) *The partial function* $\mu : (\underline{M} \times \wp_{fin}(\underline{M})) \multimap \wp(\underline{M} \times [(X \cup \underline{X}) \to \underline{M}])$ *is defined in Table 10.2.*

The case of d being a binding variable $?x$ is the basic case for the symbolic analysis, and it simply consists of assigning the current value of κ to the symbolic variable.

- If d is an atomic non-symbolic message that can be deduced from (actually, belongs to) κ then μ returns d together with the empty substitution.
- If d is a symbolic variable $x(\kappa')$, then it must have been previously generated by the intruder, and then $\kappa \trianglerighteq x(\kappa')$, by applying the last rule in the definition of $\trianglerighteq$. It is however necessary, according to Definition 10.1.4, to restrict $x(\kappa')$ to $\hat{x}(\kappa')$.
- If d is the pair e, f then μ is firstly applied to e', and κ yielding a set of possible results. For each solution (e', σ_e) in $\mu(e, \kappa)$, the substitution σ_e is propagated to f and κ, thus yielding the argument of μ which determines the possible pairs (f', σ_f). The combinations of these results together with the composition of their substitutions are returned.

  Note that the order in which e, f are visited is dictated by the binding mechanism adopted for cIP (see Definition 9.4.2).

$$\nu(m,d) = \begin{cases} \{[^{m}/_{x}]\}, & d = ?x \\ \{\varepsilon\}, & d = m \in N \cup K \\ \{\sigma_e\sigma_f : \sigma_e \in \nu(e,e') \wedge \sigma_f \in \nu(f\sigma_e, f'\sigma_e)\}, & m = e,f \ \wedge \ d = e',f' \\ \nu(e,e') & m = \{e\}_\lambda \ \wedge \ d = \{e'\}_{\lambda^-} \\ \{[^{\hat{x}(\overline{\kappa}),\hat{x}(\overline{\kappa})}/_{x(\kappa),y(\kappa')}]\}, & m = x(\kappa) \ \wedge \ d = y(\kappa') \ \wedge \ \bar{\kappa} = \kappa \sqcap \kappa' \neq \emptyset \\ \{([^{m'}/_{y_i(\kappa')}])\sigma : (m',\sigma) \in \mu(m,\kappa')\} & m \in \underline{M} \setminus \underline{X} \ \wedge \ d = y(\kappa') \\ \{([^{n}/_{x(\kappa')}])\sigma : (n,\sigma) \in \mu(\kappa',d)\} & m = x(\kappa') \ \wedge \ d \in \underline{M} \setminus \underline{X} \\ \emptyset, & \text{otherwise} \end{cases}$$

Table 10.3: Definition of ν

- If $d = \{e\}_{\lambda^-}$ is a cryptogram and the intruder owns the decryption key λ (i.e. $\lambda \in \kappa$), the problem reverts to producing the proper messages that must be encrypted. On the other hand, it must also be considered the case that κ contains cryptograms $\{e'\}_\lambda$ properly encrypted with λ. In this case, e' it must be checked whether an appropriate σ, that makes $\{e'\}_\lambda$ and $d = \{e\}_{\lambda^-}$ to match, exists or not. This is done by the auxiliary function ν, defined in Definition 10.2.2.

When none of the previous cases applies, the function μ returns the empty set of substitutions. The function ν, given a symbolic message m and a datum d, returns, if any, a set of possible substitutions σ such that $m\sigma \simeq d\sigma$.

Function μ and ν are mutually recursive. The function μ calls the function ν to verify if $\{e\}_\lambda$ and $\{e'\}_{\lambda^-}$ can match under an appropriate substitution. The function ν calls μ to verify whether a symbolic variable (contained in $\{e\}_\lambda$) may generate a message matching another given message (which occurs in $\{e'\}_{\lambda^-}$).

Definition 10.2.2 (Checking substitutions) *Given two sets of messages κ' and κ'', we let $\kappa' \sqcap \kappa''$ be the set $\{m \in \kappa' \cup \kappa'' : \kappa' \trianglerighteq m \wedge \kappa'' \trianglerighteq m\}$. If $m \in \underline{M}$ is a message and $d \in \underline{M}$ is a datum, then partial function $\nu : (\underline{M} \times \underline{M}) \multimap [(X \cup \underline{X}) \to \underline{M}]$ is defined in Table 10.3.*

Most of the other cases are straightforward, hence we report the non trivial ones. The case where d (or m) is a symbolic variable $y(\kappa')$ ($x(\kappa')$) needs some explanations. If m is a symbolic message not in $\underline{X}$ and $d = y(\kappa')$, it is necessary to check if κ' can generate a message that matches m. This is obviously done by $\mu(\kappa', m)$ (similarly for the case $m = x(\kappa')$ and $d \in \underline{M} \setminus \underline{X}$). Note that at each recursive call of μ and ν,

the complexity of the argument decreases. If both m and d are (possibly different) symbolic variables, then they can symbolically match provided that they represent a common subset of messages that both of them can derive. The set $\overline{k}$ represents the maximal set of messages that can be derived both from κ and κ', so that none of the possible concrete choices is lost. Due to dynamic evolution of κ, the intersection of the derivable messages is not a function of the intersection of the sets. Consider, for example, $\kappa = \{\{no_1\}_k\}$ and $\kappa' = \{no_1, k\}$: The intersection is empty, but both sets can derive $\{no_1\}_k$. The definition of $\overline{\kappa}$ encompasses this case. Symbolic variables should hence be mapped into the same symbolic marked variable.

Proposition 10.2.1 *If $\kappa \subseteq \underline{M}$ is finite $\mu(d, \kappa)$ is finite, for any datum d.*

PROOF. The proof trivially follows by induction on the structure of d, if we consider that

a. recursive calls in Table 10.2 are done on terms having a decreasing complexity

b. the base cases always yield finite sets.

□

Example 10.2.1 *Let κ be of the form $\{k,\ \{no_1\}_{B_2^-},\ no_1,\ \{j_3(\{A_3^+, \{A_3^+\}_k\})\}_{C_4^+}\}$. We now show how an appropriate message m, matching input datum d, may be derived from κ.*

1. *$d = z_5(\{\{no_1\}_k, \{no_1\}_{B_2^-}\})$, is a symbolic variable containing the two messages: $\{no_1\}_k$ and $\{no_1\}_{B_2^-}$. The first one can be derived in one step, while the second one directly belongs to κ, hence, by applying the last rule for $\trianglerighteq$, $\kappa \trianglerighteq z_5(\kappa')$. Hence, $\mu(n, \kappa) = (n, z_5(\kappa') \mapsto \hat{z}_5(\kappa'))$.*

2. *$d = \{w_2(\{k, A_3^+\})\}_{C_4^-}$, is a cryptogram containing a symbolic variable. The knowledge κ contains the cryptogram $\{j_3(\{A_3^+, \{A_3^+\}_k\})\}_{C_4^+}$. Then $\mu(d, \kappa)$ reverts to $\nu(j_3(\{A_3^+, \{A_3^+\}_k\}), w_2(\{k, A_3^+\}))$.*

   This is the case of two symbolic variables. Their "intersection" is the set $\overline{\kappa} = \{A_3^+, \{A_3^+\}_k\}$ which does not contains k, since $\{A_3^+, \{A_3^+\}_k\} \not\trianglerighteq k$. It follows that the two symbolic variables are mapped to $\hat{w}_2(\overline{\kappa})$:

$$\nu(j_3(\{A_3^+, \{A_3^+\}_k\}), w_2(\{k, A_3^+\})) = [^{\hat{w}_2(\overline{\kappa}), \hat{w}_2(\overline{\kappa})}/_{w_2(\_), j_3(\_)}].$$

   Then $\mu(n, \kappa) = (\{j_3(\{A_3^+, \{A_3^+\}_k\})\}_{C_4^+}, [^{\hat{w}_2(\overline{\kappa}), \hat{w}_2(\overline{\kappa})}/_{w_2(\_), j_3(\_)}]$. And again $\kappa \trianglerighteq m$ and $m\sigma \simeq d\sigma$.

Note that each choice for m respects the property that any possible symbolic variable leads to a correct *concrete communication.*

Lemma 10.2.1 *Let $\kappa \in \wp(\underline{M})$ and let σ be symbolic substitution such that κ covers any message in $dom(\sigma) \cup cod(\sigma)$, then, if $\kappa\sigma \trianglerighteq m$ then $\partial(\kappa) \trianglerighteq m$.*

PROOF. By hypotheses any $x(\kappa')$ is replaced in κ with a message m' that can be generated by κ. Indeed, it must be the case that $\kappa' \trianglerighteq m'$, hence, $\kappa \trianglerighteq m'$ because κ covers κ'. This implies that $\kappa\sigma$ cannot built m using sub-messages that cannot be built by κ. □

Observation 10.2.1 *Lemma 10.2.1 implies that if $(m, \sigma) \in \mu(d, \kappa)$, then substitution cannot be "enlarged". Namely there is no substitution σ' such that $m\sigma' \simeq d\sigma'$ and the knowledge of any symbolic variable of σ' covers the knowledge associated to it through σ. In some sense, μ determines "the most general" unifying substitution among m and d.*

Lemma 10.2.2 *Let $\kappa \in \wp(\underline{M})$ and let d be a datum such that κ covers d then, for all $(m, \sigma) \in \mu(d, \kappa)$, κ covers m and κ covers any message in $dom(\sigma) \cup cod(\sigma)$.*

PROOF. By inspecting Table 10.2 and Table 10.3 it is easy to see that all symbolic variables introduced in the result of μ are obtained by κ or by a set κ' already occurring in d. By hypothesis, κ' and d are covered by κ. □

It is necessary to guarantee that μ takes into account *all* the possible message-substitutions that can be derived from κ. The point is that the symbolic semantics that must consider all the possible evolutions that can concretely be performed in the non-symbolic semantics.

Proposition 10.2.2 *Let d be a datum and $\kappa \in \wp(\underline{M})$ cover d. If $(m, \sigma) \in \mu(d, \kappa)$ then $d\sigma \simeq m\sigma$ and $\partial(\kappa) \trianglerighteq m$.*

PROOF. The proof proceeds by induction on the structure of d.

By definition of substitution and $\trianglerighteq$, the proposition holds in the first three cases (the base cases).

Let d be the pair e, f. Recall that binding variables have at most one occurrence and bind all the remaining occurrences We first compute $\mu(e, \kappa)$ and then apply the substitutions to f and κ to compute $\mu(f\sigma_e, \kappa\sigma_e)$. If $\mu(e, \kappa) = \emptyset$ then it is not possible to derive a matching message for d. Hence, let us assume that $(e', \sigma_e) \in \mu(e, \kappa)$. By inductive hypothesis, $e'\sigma_e \simeq e\sigma_e$ and $\partial(\kappa) \trianglerighteq e'$ and, similarly, $f'\sigma_f \simeq f\sigma_e\sigma_f$ and $\partial(\kappa)\sigma_e \trianglerighteq f'$. We can apply Lemmas 10.2.1 and 10.2.2 (because $\partial(\kappa\sigma) = \partial(\kappa)\sigma$ if κ covers σ). $\kappa \trianglerighteq e', f'$ and we have

$$d\sigma_e\sigma_f \simeq (e\sigma_e\sigma_f), (f\sigma_e\sigma_f) \simeq ((e'\sigma_e)\sigma_f), ((f'\sigma_e)\sigma_f) \simeq e', f'(\sigma_e\sigma_f), \tag{10.1}$$

where symbolic matchings (10.1) hold by the usual substitution properties (in particular, $t(\sigma\sigma') = (t\sigma)\sigma'$).

Let d be the cryptogram $\{e\}_{\lambda^{-1}}$; we split the generation of a message and the corresponding matching substitution in two cases. Indeed, if the decryption key can be deduced from κ, then we simply return the matching substitution for the content of the cryptogram. By induction this proves the proposition. Now, we must consider cryptograms whose encryption key is λ and determine whether their content can be matched via a substitution with e by invoking ν. Basically, function ν works as μ but without trying to manipulate its second argument, hence, by induction, for any $\sigma_e \in \nu(e, e')$, $e\sigma_e \simeq e'\sigma_e$ and we obtain the thesis. □

Proposition 10.2.3 *Given an input datum d, a message m and a set $\kappa \in \wp_{fin}(\underline{M})$. If $\partial(\kappa) \trianglerighteq m$ and $m\sigma \simeq d\sigma$ for a symbolic substitution of the variables in d, σ, covered by κ, then there is $(\underline{m}, \underline{\sigma}) \in \mu(d, \kappa)$ such that for any mapping $[^{m'}/_{x(\kappa')}]$ in σ, a substitution ρ exists that replaces symbolic variables in m' with larger associated knowledge. Moreover, $[^{m'\rho}/_{x(\kappa')}]$ is a substitution in $\underline{\sigma}$ (and analogously for mappings $\sigma : x \mapsto m'$).*

Proof. We proceed by induction on the structure of d. If $d = ?x$ then $(x, [^{x(\kappa)}/_{x}]) \in \mu(d, \kappa)$ by definition and, by hypothesis, $d\sigma = m'$ and $\partial(\kappa) \trianglerighteq m'$. Clearly, κ covers m'. If $d \in N \cup K$ then $d\sigma = d$ for any substitution, hence, by hypothesis, $m\sigma = d$. If $m \simeq d$ then we have that $\partial(\kappa) \trianglerighteq d$ and, by definition, $(d, \varepsilon) \in \mu(d, \kappa)$ otherwise, m is a variable such that $\sigma : m \mapsto d$ and $\partial(\kappa) \trianglerighteq d$ because κ covers σ. When $d = x(\kappa')$, $(x(\kappa'), [^{\hat{x}(\kappa')}/_{x(\kappa')}]) \in \mu(d, \kappa)$ because κ covers σ whose domain contains $(x(\kappa')$.
The other cases follows by inductive hypothesis. □

Proposition 10.2.3 states that μ takes into account any pair (m, σ) that can be used in the hypothesis of the inference rule ($\underline{in}$) in Table 10.1. Intuitively, this means that messages matching a datum d either have a structure similar to d or, at some level they are replaced by symbolic variables. Since μ always introduces symbolic variables that cover any other variable that can be generated by κ, we can use such messages for considering all the possible symbolic ($\underline{in}$) transitions. This corresponds to saying that, given κ, μ computes the "most general unifying" substitution between m and d. This result is also important for relating symbolic and concrete traces as will be shown in next section.

As a final remark the above construction is quite similar to the well-known unification algorithm of [118]. This also suggests that the complexity for computing μ and ν is comparable to the classical unification algorithm.

10.3 Intruder construction

We show how terminating symbolic traces provide all the information necessary for constructing an intruder that interacts with the principals of a protocol and tries to

cheat them.

Protocol analysis can be conducted in three phases:

1. a cIP bunch of principals is derived from the informal specification;
2. a security property is specified by means of a $\mathcal{PL}$ formula ϕ;
3. a model for $\neg\phi$ is searched for.

Step 1 is a human activity even if cIP features are tailored for driving this phase. Similarly, step 2 is a phase that requires human intervention because also the properties are usually informally stated. However, $\mathcal{PL}$ logic allows one to reason about variables and rôles described by the principals in a way that is natural enough for the formalization. Step 3, instead, is completely algorithmic. Essentially, starting from a context that satisfies the initial conditions of the protocol and, by applying the inference rules of the context semantics, one constructs traces which end in a state where the intruder knowledge and the set of bindings are a model for $\neg\phi$.

We consider only traces starting from *initial* states and ending in *final* states.

Definition 10.3.1 (Initial and final states) *A state $\langle \mathcal{C}, \chi, \kappa \rangle$ is*

- initial *if, and only if, $\mathcal{C} = \emptyset$, χ is the empty substitution and $\kappa \in \wp_{fin}(M)$;*
- final *if, and only if, $\mathcal{C}$ contains only principals of the form $()[\mathbf{0}]$, χ is a symbolic substitution and $\kappa \in \wp_{fin}(\underline{M})$.*

An initial state represents a context that still does not contain any participant. The intruder may be equipped with a finite (non-symbolic) knowledge. Usually, this knowledge contains (private) names, public keys or other information learned by the intruder in previous interaction with some principal. A final state represents the "goal" of the intruder which is to drive all the participants that took part to the protocol in a terminal state where the security property is violated.

Definition 10.3.2 (Symbolic traces) *A* symbolic trace *is a sequence*

$$\underline{T} \;=\; \Sigma_0.\alpha_1.\ldots.\alpha_n.\Sigma_n$$

where Σ_0 is an initial state and $\Sigma_{i-1} \overset{\alpha_i}{\rightsquigarrow} \Sigma_i$ $(1 \leq i \leq n)$. $\underline{T}$ is terminating *if Σ_n is a final state.*

Example 10.3.1 *Let us again consider the principals of the Needham-Schroeder protocol:*

$$\begin{array}{rcl} A & \triangleq & (y)[out(\{na, A\}_{y^+}).in(\{na, ?u\}_{A^-}).out(\{u\}_{y^+})] \\ B & \triangleq & ()[in(\{?x, ?z\}_{B^-}).out(\{x, nb\}_{z^+}).in(\{nb\}_{B^-})] \end{array}$$

We let $\underline{T}_{NS}$ be the trace below (prefix actions triggering transitions are evidentiated in boldface characters)

$$\langle \emptyset, \varepsilon, \emptyset \rangle$$

$$\downarrow j\ A_1\ \varepsilon$$

$$\langle \{(y_1)[\mathbf{out}(\{\mathbf{na_1}, \mathbf{A_1}\}_{\mathbf{y_1^+}}).in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{y_1^+})]\}, \varepsilon, \{A_1\} \rangle$$

$$\downarrow o\ \{na_1, A_1\}_{y_1^+}$$

$$\langle \{(y_1)[in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{y_1^+})]\}, \varepsilon, \{A_1, \{na_1, A_1\}_{y_1^+}\} \rangle$$

$$\downarrow j\ B_2\ [^{B_2}/_{y_1}]$$

$$\langle \left\{ \begin{array}{c} ()[in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{B_2^+})], \\ ()[\mathbf{in}(\{?\mathbf{x_2}, ?\mathbf{z_2}\}_{\mathbf{B_2^-}}).out(\{x_2, nb_2\}_{z_2^+}).in(\{nb_2\}_{B_2^-})] \end{array} \right\}, [^{B_2}/_{y_1}], \{A_1, B_2, \{na_1, A_1\}_{B_2^+}\} \rangle$$

$$\downarrow i\ \{x_2(\kappa), A_1^+\}_{B_2^+}$$

$$\langle \left\{ \begin{array}{c} ()[in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{B_2^+})], \\ ()[out(\{x_2(\kappa), nb_2\}_{A_1^+}).in(\{nb_2\}_{B_2^-})] \end{array} \right\}, [^{B_2, x_2(\kappa), A_1}/_{y_1, x_2, z_2}], \kappa \rangle$$

where $\kappa = \{A_1, B_2, \{na_1, A_1\}_{y_1^+}\}$.
Premises of the rule (in) requires that the chosen substitution must be a valid symbolic substitution, therefore, while x_2 is mapped to the symbolic variable $x_2(\kappa)$, z_2 must be mapped to a key in κ and, in this case A_1^+ is chosen.

Given a protocol and a security property ϕ, a symbolic intruder is a symbolic trace that, once "concretization" has taken place (i.e. all its symbolic variable are substituted with suitable concrete messages in M), gives rise to a trace of the concrete semantics (according to Definition 9.4.10) such that bindings and knowledge of the last state in the trace are model for $\neg\phi$.

Definition 10.3.3 (Concretizing substitution) *Given a symbolic substitution χ, a* concretizing substitution for χ *is a substitution $\rho : \underline{X} \to M$ such that, for any symbolic variable $x(\kappa)$ that occurs in a message of $cod(\chi)$, no symbolic variable occurs in messages of $\kappa\rho$ and $\kappa\rho \rhd x(\kappa)\rho$.*

A concretizing substitution replaces all symbolic variables $x(\kappa)$ with concrete data that can be derived from κ. If $\underline{T}$ is a symbolic trace whose last state has bindings χ and ρ is a concretizing substitution of χ, then we say that ρ is a *concretizing substitution for* $\underline{T}$.

Given a state $\Sigma = \langle \mathcal{C}, \chi, \kappa \rangle$ and a substitution ρ, we write $\Sigma\rho$ to denote the application of ρ to Σ, namely the state $\langle \mathcal{C}\rho, \chi; \rho, \kappa\rho \rangle$. Similarly, given a trace $\underline{T}$, $\underline{T}\rho$ is the trace obtained by applying ρ to each state and each label in $\underline{T}$.

Example 10.3.2 *A concretizing substitution for trace $\underline{T}_{NS}$ of Example 10.3.1 is a substitution that maps $x_2(\kappa)$ to a value deducible from κ. For instance, we can concretize $\underline{T}_{NS}$ by applying the substitution $[^{A_1,B_2^+}/_{x_2(\kappa)}]$ and obtain the following trace*

$$\langle \emptyset, \varepsilon, \emptyset \rangle$$

$$\downarrow j\ A_1\ \gamma$$

$$\langle \{(y_1)[out(\{na_1, A_1\}_{y_1^+}).in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{y_1^+})]\}, \varepsilon, \{A_1\} \rangle$$

$$\downarrow o\ \{na_1,A_1\}_{y_1^+}$$

$$\langle \{(y_1)[in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{y_1^+})]\}, \varepsilon, \{A_1, \{na_1, A_1\}_{y_1^+}\} \rangle$$

$$\downarrow j\ B_2\ [^{B_2}/_{y_1}]$$

$$\langle \left\{ \begin{array}{c} ()[in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{B_2^+})], \\ ()[in(\{?x_2, ?z_2\}_{B_2^-}).out(\{x_2, nb_2\}_{z_2^+}).in(\{nb_2\}_{B_2^-})] \end{array} \right\}, [^{B_2}/_{y_1}], \{A_1, \{na_1, A_1\}_{y_1^+}\} \rangle$$

$$\downarrow i\ \{A_1,B_2^+,A_1,B_2^+\}_{B_2^+}$$

$$\langle \left\{ \begin{array}{c} ()[in(\{na_1, ?u_1\}_{A_1^-}).out(\{u_1\}_{B_2^+})], \\ ()[out(\{A_1, B_2^+, nb_2\}_{A_1^+}).in(\{nb_2\}_{B_2^-})] \end{array} \right\}, [^{B_2,A_1,B_2^+,A_1}/_{y_1,x_2,z_2}], \kappa \rangle$$

where κ is the same of Example 10.3.1.

Symbolic semantics is coherent with respect to the concrete semantics. Indeed, any symbolic trace can be concretized to a non-symbolic trace as stated by the following theorem.

Theorem 10.3.1 *If $\underline{T}$ is a symbolic trace, for all ρ concretizing substitutions for $\underline{T}$, the states of $\underline{T}\rho$ form a trace deducible from the transition system in Definition 9.4.10.*

Proof. Let $\underline{T} = \Sigma_0.\alpha_1.\ldots.\alpha_n.\Sigma_n = \langle \mathcal{C}, \chi, \kappa \rangle$ be a symbolic trace. The proof easily follows by induction on the length of $\underline{T}$, n. If $n = 0$ there is nothing to prove because Σ_0 is a concrete state, by definition. Let assume that the theorem holds for any trace on length n and let us consider the trace $\underline{T}\alpha_{n+1}\Sigma_{n+1} = \langle \mathcal{C}', \chi', \kappa' \rangle$. By construction, χ' either is obtained by extending χ with substitutions of new symbolic variable or by refining variable already in $cod(\chi)$. Hence, if we consider a concretizing substitution for χ', we can observe that it also is a concretizing substitution for χ, therefore (the state of) $\underline{T}\rho$ is a trace in the concrete semantics of cIP by inductive hypothesis. We proceed by case analysis on α_{n+1}

- $\boxed{\alpha_{n+1} = j\ A_i\ \gamma}$: then $\Sigma_{n+1} = \langle join(A_i, \gamma, \mathcal{C}), \chi\gamma, \kappa\gamma\rangle$. Hence,

$$\Sigma_{n+1}\rho = \langle join(A_i, \gamma, \mathcal{C}\rho), \chi\rho\gamma, \kappa\rho\gamma\rangle$$

  since γ does not introduce symbolic variables and, therefore, $\gamma\rho = \rho\gamma$. Clearly, $\Sigma_n\rho \mapsto \Sigma_{n+1}\rho$ can be obtained by applying the $(join)$ rule.

- $\boxed{\alpha_{n+1} = o\ m\sigma}$: in this case there must be a principal in $\mathcal{C}$ that performs an $in(d)$ action and σ is valid substitution. Without loss of generality (for Proposition 10.2.2 and the results in Section 10.1) we can assume that there is a substitution σ' such that $(m, \sigma') \in \mu(d, \kappa)$ and σ is obtained by σ' by concretizing some symbolic variable to keys (to obtain a valid symbolic substitution). This implies that $m\sigma\rho$ is deducible from $\kappa\rho$ and matches $d\rho$. Finally, $\mathcal{C}\rho$ contains the concretization of the principal performing the $in(d)$ action, therefore, we can apply rule (in) and obtain the thesis.

- $\boxed{\alpha_{n+1} = i\ m}$: is dealt similarly to the previous case.

□

Theorem 10.3.2 *If $T = \langle\emptyset, \varepsilon, \kappa_0\rangle \mapsto \ldots \langle\mathcal{C}_n, \chi_n, \kappa_n\rangle$ is a concrete trace then there are*

1. *a symbolic trace $\underline{T}$ reaching a state $\langle\mathcal{C}', \chi', \kappa'\rangle$*
2. *and a concretizing substitution ρ for χ'*

such that, for any $i = 1, ..., n$ the i-th state in T is the concretization via ρ of the i-th state of $\underline{T}$.

Proof. The proof follows the same pattern of the proof of Theorem 10.3.1.

The cases of rules $(join)$ and (out) are straightforward.

The interesting cases are the transitions obtained by applying rule (in). By Proposition 10.2.3 (and Observation 10.2.1) we can prove the thesis by *absurdum*. Let assume that it is not possible to mimic a concrete input action. This means that function μ returns the empty set. This contradicts the fact that in the concrete semantics we could derive a message from the intruder knowledge matching the input datum. By inductive hypothesis, such knowledge can be obtained by concretizing the corresponding symbolic one. Hence such concretization would also generate the message chosen in the transition and, by Proposition 10.2.3, this is not possible. □

By constructing symbolic traces, it is possible to collect symbolic bindings and intruder's knowledge that the protocol execution can determine. Each terminating trace represents a potential model for the formula to be verified.

Definition 10.3.4 (Symbolic intruder) *Let* $\underline{T} = \Sigma_0\alpha_1 \ldots \alpha_n\Sigma_n$ *be a symbolic trace and let* ϕ *be a* $\mathcal{PL}$*-formula. Trace* $\underline{T}$ *is a* symbolic intruder for ϕ *if, assuming* $\Sigma_n = \langle \mathcal{C}, \chi, \kappa \rangle$*, a concretizing substitution* ρ *exists such that* $\kappa\rho \models_{\chi\rho} \phi$*.*

10.4 Symbolic models

We already pointed out that symbolic analysis makes finite the number of possible messages that can be exchanged since the number of principal instances is fixed and finite. Hence the space of the reachable states is finite, too.

We provide a notion of symbolic model $\kappa \models\!\!\models_\chi \phi$ for a knowledge κ that may contain symbolic messages, a symbolic substitution χ and a closed $\mathcal{PL}$ formula ϕ. Given a formula ϕ we show how it can guide in concretizing χ by appropriately restricting symbolic variables and preserving satisfiability of ϕ. The impossibility of concretizing χ in this way implies that ϕ is not satisfiable.

Definition 10.4.1 (Symbolic models) *Let* $\chi : X \cup \underline{X} \to \underline{M}$ *be a symbolic substitution and let* $\kappa \in \wp_{fin}(\underline{M})$*. The pair* $\langle \kappa, \chi \rangle$ *is a* symbolic model for a closed formula ϕ *(written* $\kappa \models\!\!\models_\chi \phi$*) if, and only if, a concretizing substitution* ρ *for* χ *exists and* $\kappa\rho \models_{\chi\rho} \phi$ *holds.*

We are interested in defining models of formulae expressing security properties of protocols sessions. Indeed, we aim at showing how to use formulae themselves to further constrain symbolic values introduced during protocol execution. Informally, given a formula ϕ, once a symbolic substitution and a finite set of symbolic messages κ have been obtained by the execution of the protocol, we can proceed as follows:

1. The formula is transformed in an equivalent formula where
   - the quantifiers have been eliminated;
   - the $\neg$ operator is pushed as far as possible inside the formula;
   - finally, the result is transformed in a disjunction of conjuncts.
2. Disjuncts are searched to determine a symbolic substitution that further constraints variables in the conjuncts.

The procedure sketched above can also yield the undefined substitution or a contradictory set of inequalities (e.g. $x(\kappa_1) \neq x(\kappa_2)$). In such case the initial κ and χ cannot be a symbolic model for the formula ϕ.

Step 1 is immediate. Indeed, universal and existential quantifiers can be replaced by a finite conjunction and disjunction respectively by instantiating the session variables with the principal names appearing in the intruder knowledge. The remaining transformations are ensured by the De Morgan laws.

Definition 10.4.2 (Normal form formulae) *A $\mathcal{PL}$ formula ϕ is in* normal form *if*

$$\phi = \bigvee_{i \in 1,\ldots,u} (\psi_{i,1} \wedge \ldots \wedge \psi_{i,j_i})$$

$\psi_{i,j}$'s are negative or positive atoms.

Proposition 10.4.1 *For any $\mathcal{PL}$ formula ϕ, $\kappa \in \wp_{fin}(M)$ and substitution χ, a normal form ψ exists and it is such that $\kappa \models_\chi \phi \iff \kappa \models_\chi \psi$.*

Proof. The proof easily follows by induction on the structure of ϕ and the definition of models of $\mathcal{PL}$ formulae. □

Proposition 1 suggests that model checking $\mathcal{PL}$ formulae can be reduced to to model check formulae that are in normal form. A further simplification is to consider only conjuncts atoms because it suffices to find a model for one of the disjunction to obtain a model for the whole formula in normal form.

The next section tackles the problem of deciding whether a set of symbolic messages and a symbolic substitution can be refined for satisfying a conjunction of atoms.

10.4.1 Constraining atoms

Given a symbolic knowledge, a symbolic substitution, and a formula ϕ that is a conjunction of atoms we show how positive atoms can be exploited for determining a substitution (if any) that "refines" the symbolic variables occurring in the substitution. Negative atoms are used to establish the inequalities that must be granted in the models of ϕ. Once the symbolic substitution has been refined from positive atoms we can "apply" it to negative atoms and determine whether they are satisfied or not.

Informally, the constraints from positive atoms are computed by mimicking the unification algorithm of logic programming [118]: A conjunction of positive atoms is considered as a set from which an atom is non-deterministically selected and removed in order to compute a substitution that must be propagated to the remaining atoms until the set is not empty. The resulting substitution, when concretized, is granted to satisfy all the atoms.

Let us consider a formula ϕ that is a conjunction of positive atoms. We determine (if possible) a substitution that constraints the symbolic variables of the atoms of ϕ by preserving satisfiability. Hereafter, we consider ϕ to be a set of atoms instead of as a conjunct of atoms. This is a more suitable representation for describing the algorithm that computes the *refinement substitution* from ϕ.

Definition 10.4.3 (Constraining equalities) *Given $\kappa \in \wp_{fin}(\underline{M})$ and a symbolic substitution χ, the* refinement substitution Ψ *is defined in Table 10.4 where m^{-1} is*

$$\Psi_{\kappa,\chi}(\phi) = \begin{cases} \Psi_{\kappa\sigma,\chi\sigma}(\phi'), & \phi = \phi' \cup \{\delta_1 = \delta_2\} \ \wedge \ \delta_h\chi = x^{(h)}(\kappa^{(h)}),\ (h = 1,2) \ \wedge \\ & \bar{\kappa} = \kappa^{(1)} \sqcap \kappa^{(2)} \neq \emptyset \ \wedge \ i \leq j \ \wedge \ \sigma = [{}^{x_i(\bar{\kappa}),x_i(\bar{\kappa})}/_{x_i(\kappa'),y_j(\kappa'')}] \\ \Psi_{\kappa\sigma,\chi\sigma}(\phi'), & \phi = \phi' \cup \{\delta_1 = \delta_2\} \ \wedge \ \delta_1\chi = x(\kappa') \ \wedge \ \delta_2\chi \in \underline{M} \setminus \underline{X} \ \wedge \\ & x \text{ does not occur in } \delta_2\chi \ \wedge \\ & \sigma \text{ in } \mu((\delta\chi)^{-1},\kappa') \ \wedge \ \sigma = \sigma'[{}^{\delta\chi\sigma'}/_{x(\kappa')}] \\ \Psi_{\kappa,\chi}(\phi'), & \phi = \phi' \cup \{\delta = \delta\} \ \wedge \ \delta \in N \cup K \\ \Psi_{\kappa\sigma,\chi\sigma}(\phi'), & \phi = \phi' \cup \{\delta \in \mathfrak{K}\} \ \wedge \ \sigma' \text{ in } \mu((\delta\chi)^{-1},\kappa) \ \wedge \ \sigma = \sigma'[{}^{\delta\chi\sigma'}/_{x(\kappa')}] \\ \chi, & \phi = \emptyset \\ \bot, & \text{otherwise} \end{cases}$$

Table 10.4: Constraint refinement function

the message obtained from replacing all encryption keys λ in message m with its inverse key λ^{-1}.

Given a set of atoms ϕ, $\Psi_{\kappa,\chi}$ choses an atom from ϕ and determines a substitution that, if different from $\bot$, is propagated on κ and χ in the recursive invocations of Ψ. If the returned substitution is $\bot$ then $\Psi_{\kappa,\chi}$ returns $\bot$ too. Table 10.4 defines $\Psi_{\kappa,\chi}(\phi)$ by induction on the set ϕ. It basically checks, for each atom, whether symbolic variables can be replaced by messages to either unify right-hand- and left-hand-side of equalities, or determine the values that make a message belong to the intruder knowledge.

Let us comment on the clauses of Table 10.4. If $\delta_1 = \delta_2$ is an equality of ϕ, then three cases are possible when we apply χ to the equality:

1. two symbolic variables are equated, e.g. $x^{(1)}(\kappa^{(1)}) = x^{(2)}(\kappa^{(2)})$,
2. a variable and a message are equated, e.g. $x(\kappa') = m$, where $m = \delta_2\chi \in \underline{M}$,
3. the equation is a tautology of the form $n = n$.

The above cases corresponds to the first three cases of Table 10.4. Case *1* says that $\Psi_{\kappa,\chi}(\phi)$ first checks if $\bar{\kappa}$ empty or not. In the former case $\bot$ is returned, otherwise substitution σ determined as in Table 10.4 is propagated to κ and χ and Ψ handles the remaining equalities. Condition $i \leq j$ is imposed only to determine a canonical substitution σ. Case *2* is similar because it Ψ checks if κ' can generate the message and propagates the substitution σ. Case *3* is trivial, the tautology is simply removed from the set of conjunctions. When ϕ is empty $\Psi_{\kappa,\chi}$ terminates returning χ. All the other possibilities correspond to the case where an atom cannot be satisfied, hence Ψ returns $\bot$.

Once a substitution has been refined exploiting positive atoms of a conjunction of atoms, we can use negative atoms of the conjunction for testing whether the

refined model is a model for the whole conjunction. Indeed, if one of the atoms is not satisfied by the model it cannot satisfy the whole formula. When all negative atoms are satisfied we have found a model that can be concretized to obtain a non-symbolic model of the initial conjunction. Moreover, this test is simpler than the refinement mechanism of positive atoms because we must only consider three cases:

- $x_1(\kappa) \neq m$;
- $x_i(\kappa_1) \neq y_j(\kappa_2)$;
- $m \notin \mathfrak{K}$.

By observing that a non-empty set of messages can generate infinite messages, we can immediately derive that the first case is always true, and the second is true when x_i and y_j are different variables, while $x_i(\kappa_1) \neq x_i(\kappa_2)$ is a contradiction. The last case means that $\kappa \not\trianglerighteq m$ must be checked.

10.5 Concluding Remarks

The number of verification techniques that have been proposed for the formal analysis of systems is quite large and it makes not possible to list all of them here. We cite a small subset of these approaches that are somehow related to our proposal. Process calculi have been intensively used in protocol analysis [79, 80, 115, 117, 163, 164, 160, 2, 22, 23]. Other approaches rely on inductive theorem proving [147], and on logic programming [122]. All these approaches made simplifying hypothesis on the behaviour of the intruder to bound the state space in order to apply standard finite state verification techniques

In particular our approach is very close to [22] where a symbolic semantics have been successfully specified for the spi calculus [2]. In [22, 23] it is shown how the symbolic approach sensibly reduces state space exploration with respect the non-symbolic model checkers. Our framework generates a bigger state space compared to [23, 22] because we avoid generating symbolic traces which have no concrete counterpart, while [23, 22], after having generated a symbolic trace, must filter concretions that are really possible in the concrete semantics. Despite of state space dimension, our framework is more efficient because we perform model checking of security properties only in the final states of the traces.

The direct handling of multiple sessions is the distinguished feature of our approach with respect to the other process calculi for security. The relevance of multi-session attacks has been emphasizes in [129]. Some results [116] had proved that, in particular cases, it is possible to impose sufficient conditions that allow one to limit the analysis to "small system", i.e. contexts where few sessions are present. However, those results are not general and only hold for secrecy properties.

Another approach are *strand spaces* [69, 68], where a participant of a protocol is modeled as a finite strand of nodes representing input or output actions; each input node may receive its value from a single output node. A principal cannot have non-deterministic behaviour. A similar idea has been followed in [44]. An interesting connection with cIP and stands can be outlined if we restrict the syntax of the language avoiding + and | operators of behavioral expressions. Indeed, this would allows one to write only "single line" protocols, namely protocols where principals can only be sequential processes as much like the strand space approach. Another similarities is the pattern matching mechanism in communications. We conjecture that there is an isomorphism between the computations of cIP processes and the related strand spaces. This will be studied in future works.

Several logical formalisms have been proposed to specify properties of security protocols. Here, we mention the BAN logic [31], the *correspondence* relations of Dolev and Yao [66]. In the BAN logic a protocol P is *idealized* to formally state the intended meaning of message exchanged by principals. However, this is a non trivial task and requires some expertise. The idealized protocol is annotated with assertions that recall pre- and post-conditions of the Hoare Logic. Clearly, proofs specify what is true initially and after the "execution" of an idealized step. Finally, the annotated initialization is automatically checked. The Dolev and Yao correspondence relations have been adopted in [133, 101, 22]. A correspondence, written as $\alpha \hookleftarrow \beta$, asserts that, for all traces, each occurrence of an action α must be preceded by an occurrence of an action β. By exploiting the correspondence relations it is possible to state authentication properties and, with some machinery, also secrecy properties. However, a main drawback is that properties are related to the number of sessions under analysis.

Part III

Co-Algebraic Minimization of Automata

Abstract

This part of the dissertation describes Mihda, an environment for the semantic minimization of automata. The idea of semantic minimization is to collapse states of automata that are equivalent. The environment can be parameterized with respect to different classes of automata and equivalent relations between states.

We will consider the design choices and the implementation of the environment with a particular emphasis to *History Dependent Automata* which are an operational model for history dependent calculi. Mihda can be exploited for minimizing π-calculus agents considering bisimulation as the equivalence. Currently we have implemented bisimulation checking for the early strong bisimulation. However, few modules can be defined and included for employing Mihda for different semantics, e.g. late or weak semantics.

Even though we will focus on History Dependent automata, we emphasize that Mihda has been designed with the aim of being a general environment for modular minimization. In particular, we also have defined a module for specifying ordinary automata and have exploited Mihda for their semantic minimization using the usual notion of bisimilarity. We remark that Mihda can also be parameterized with respect to different notion of equivalences.

Finally, we show how Mihda can be integrated with existing verification environments via a web interface that allows one to easily write simple programs that use a web interface to the verification facilities as a normal library.

Chapter 11

Co-algebraic Verification of Mobile Process

Abstract

This chapter recaps elementary notions from category theory and co-algebras that will make more clear the presentation of our results. In particular, we point out how co-algebras can be suitably exploited for representing automata and for defining a semantic minimization algorithm. The chapter ends with a comparison between the co-algebraic approach and the classical approach to minimization.

Contents

11.1 Preliminaries

It is quite common to consider concurrent and distributed systems as *reactive*, namely as systems which are plugged and executed into an environment that can interact with them by means of some *stimuli* to which systems react. In this context the behaviour of a system can be represented as the ability of the system of reacting to a given class of stimuli. Hence, a natural question is: when two systems have equivalent behaviours? The ability to answer this question is quite important. For instance, it implies that a system S can be unplugged and substituted with a system S', provided that S' is equivalent to S, namely, provided that the rest of the environment cannot distinguish the behaviour of S' from the behaviour of S. Another reason is related to "efficiency". We can replace S with a "smaller" system S', provided that they are equivalent. Among the wide number of theories for representing systems and their behaviours that have been proposed, π-calculus and *bisimulation* equivalences probably are the most famous and applied.

A very natural and elegant way of describing transitions systems is provided by *coalgebras*, that are the dual concept of algebra. Duality between algebras and coalgebra can be precisely stated in a categorical setting. This section aims at formally reviewing elementary notions of coalgebras. Indeed, we recap only minimal notions necessary for presenting the coalgebraic version of HD-automata. The reader can skip this section if (s)he is already acquainted with the notions of category, functor and co-algebra and with the elementary (polynomial) functors over **Set**. The interested reader is referred to [103, 3] for a deeper study of coalgebras.

We first introduce the concept of *category*.

Definition 11.1.1 (Category) *A* category $\mathbf{C}$ *is class of* objects $O_{\mathbf{C}}$ *(ranged over by a, b ,...) together with a class of* arrows $A_{\mathbf{C}}$ *(ranged over by by f, g ,...) such that the following properties hold:*

- *Each arrow f has a* domain $dom(f)$ *(also called* source *and a* codomain $cod(f)$ *(also called* target*) which are objects. We write $f : a \to b$ when f is an arrow whose domain is a and whose codomain is b.*

- *Given two arrows f and g such that $cod(f) = dom(g)$, the* composition *of f and g, written $f; g$, is an arrow with domain $dom(f)$ and codomain $cod(g)$.*

- *composition is associative, namely, whenever f, g and h can be composed, $f; (g; h) = (f; g); h$.*

- *For any object a there is an* identity arrow $id_a : a \to a$. *All identity arrows enjoy the following properties:*

$$id_{dom(f)}; f = f = f; id_{cod(f)}.$$

Essentially, a category is a collection of objects having a given structure and a collection of transformations of objects that preserve the structure of objects. We avoid the details of the general theory of category and we limit our presentation to the restricted setting of sets and functions among them.

Observation 11.1.1 *The category of sets, denoted by* **Set**, *is the category having sets as objects and (total) function on sets as arrows. Domain and codomain are the domain and codomain of a function, composition of arrows is the usual function composition, while identities are the identity function. It is a simple exercise to show that* **Set** *is an instance of Definition 11.1.1.*

The most important concept from category theory that we need is *functoriality*. Informally, an operation on sets is "functorial" when it can be lifted to functions preserving function composition and identities. Functoriality is a familiar concept in many fields of computer science.

Example 11.1.1 *Let $L : A \to A^*$ be the function that associates to a set A, the set of the finite lists over A, given a function $f : A \to B$, we can define $L(f) : L(A) \to L(B)$ as the function such that*

$$L(f) : [e_1, ..., e_n] \mapsto [f(e_1), ..., f(e_n)].$$

Notice that $L(f)$ is the usual map operation on lists exploited in functional programming. It is easy to prove that L is functorial, indeed, $L(f; g) = L(f); L(g)$ (if f and g can be composed) and that $L(id_A) = id_{L(A)}$.

The following definition formalizes this concept:

Definition 11.1.2 (Functor over Set) *An* (endo-)functor $\mathcal{F}$ *over* **Set** *maps sets to sets and functions to functions such that*

- *for each function $f : A \to B$, $\mathcal{F}(f) : \mathcal{F}(A) \to \mathcal{F}(A)$;*
- *for each set A, $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$;*
- *for all composable functions $f : A \to B$ and $g : B \to C$, $\mathcal{F}(f; g) = \mathcal{F}(f); \mathcal{F}(g)$.*

Figure 11.1 gives a graphical representation of how a functor acts on objects and arrows. In particular, the figure shows how relations among objects and arrows of the starting category are maintained in the target category.

Observation 11.1.2 *The general definition of functor is given for any two categories* $\mathbf{C}$ *and* $\mathbf{C}'$. *Figure 11.1 remains the same also in the general case apart that* **Set** *is substituted by* $\mathbf{C}$ *on the left and with* $\mathbf{C}'$ *on the right of* $\mathcal{F}$.

By simply applying Definition 11.1.2 it is possible to show that the identity mapping of sets and functions, or the mapping that associates a constant set L to any set A are functors overs **Set**. Other examples of functor over sets can be given.

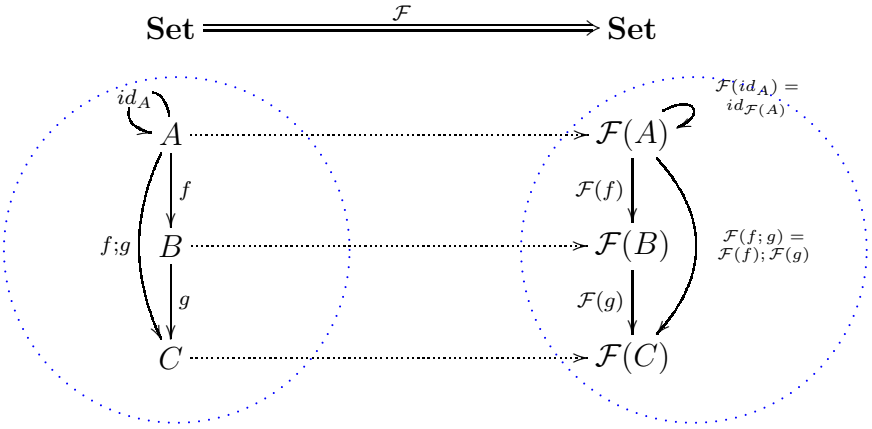


Figure 11.1: Functor over **Set**

Example 11.1.2 *This example defines two of most useful functors over* **Set**, *namely* product *and* co-product *(or disjoint union).*

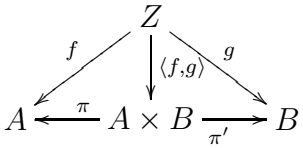
Let $A \times B$ the cartesian product of sets A and B. It is possible to define two functions $\pi : A \times B \to A$ and $\pi' : A \times B \to B$ that behaves as the projection *function, i.e.*

$$\pi : (a, b) \mapsto a \qquad \pi' : (a, b) \mapsto b.$$

Given two functions $f : Z \to A$ and $g : Z \to B$, there exists a unique pair function *$\langle f, g\rangle : Z \to A \times B$ such that the following equalities hold:*

$$\langle f, g\rangle; \pi = f \qquad \langle f, g\rangle; \pi' = g.$$

It is worth to give a graphical representation of the above relations between π, π', f, g and $\langle f, g\rangle$. More precisely, such relations express that the following diagram "commutes"



Commutativity of the diagram means that any two paths starting from the same vertex and ending in the same vertex are equal if interpreted as composition of the arrows composing the paths. Moreover, observe that $\langle \pi, \pi'\rangle = id_{A\times B}$ and that $h; \langle f, g\rangle = \langle h; f, h; g\rangle$ for any function h such that $cod(h) = Z$.
We can lift the cartesian product to functions. Indeed, if $f : A \to B$ and $f' : A' \to B'$, we can define $f \times f' : A \times A' \to B \times B'$ as the function $\langle \pi; f, \pi'; f'\rangle$, namely, the function that maps (a, a') into $(f(a), f'(a'))$. It is easy to verify that the product is functorial, in other words,the following equalities hold:

$$id_A \times id_{A'} = id_{A\times A'} \qquad (f; g) \times (f'; g') = (f \times f'); (g \times g').$$

Let $A + B$ denotes the disjoint union of sets A and B:

$$A + B \stackrel{\text{def}}{=} \{0\} \times A \cup \{1\} \times B.$$

In some sense, disjoint union is the dual of product, from which its synonym co-product *derives: Instead of projections, we can define* co-projections $\kappa : A \to A + B$ *and* $\kappa' : B \to A + B$ *which are defined by*

$$\kappa : a \mapsto (0, a) \qquad\qquad \kappa' : b \mapsto (1, b).$$

Informally, κ and κ' inject elements of A and B (respectively) into $A + B$, while, on the contrary, projections π and π' "extract" elements of A and B (resp.) from $A \times B$. Moreover, analogously to what is done for products, given functions $f : A \to Z$ and $g : B \to Z$, we can built the "co-product *function $[f, g] : A + B \to Z$ as the unique function such that $\kappa; [f, g] = f$ and $\kappa'; [f, g] = g$. It is possible to define $[f, g]$ by case:*

$$[f, g](x) = \begin{cases} f(a), \text{if } x = (0, a) \\ g(b), \text{if } x = (1, b) \end{cases}$$

Finally, we lift the co-product to functions by defining $f + g = [f; \kappa, g; \kappa']$. Observe $+$ on functions preserves identities and compositions, i.e. it is a functor.

Another functor that will be very important in defining co-algebras is the *powerset functor*.

Example 11.1.3 *Let us consider the operation $A \mapsto \wp(A)$, i.e. the function that associates to a set the set of all its subsets and, for a function $f : A \to B$, let us consider*

$$\wp(f) : \wp(A) \to \wp(B) \qquad\qquad \wp(f) : U \mapsto \{f(u) \mid u \in U\}.$$

Then, by definition,

- *$\wp(id_A)(U) = \{id_A(u) \mid u \in U\}$, for any $U \subseteq A$ hence, $\wp(id_A)(U) = U$;*
- *$\wp(f; g)(U) = \{g(f(u)) \mid u \in U\}$, for any $U \subseteq dom(f)$, hence, by definition, $\wp(f; g)(U) = \wp(g)(\wp(f)(U))$, for all $U \subseteq dom(f)$ which amounts to $\wp(f; g) = \wp(f); \wp(g)$.*

This proves that the powerset operation is functorial.

We conclude this section by claiming that the above functors are part of the so called polynomial functors that are those functors that can be obtained by combining sums, products, powerset, constant and exponentiation functors. Indeed, it is possible to prove that any combination of this functors is a functor (in fact categories and functors among them form a category) [175, 5].

11.2 Algebras and coalgebras

Before introducing coalgebras, we show how it is possible to rephrase the more familiar concept of Σ-algebra into a categorical framework. This approach also permits us to state the duality of algebras and coalgebras.

Given a signature Σ that, for the sake of simplicity we consider one sorted. We can easily define a Σ-algebra, namely a structure over a given set A that associates functions to any symbol in Σ. The only constraint being the fact that such functions must preserve arity of operations[1].

Example 11.2.1 *If* $\Sigma = \langle \mathtt{nat}, \mathtt{z} :\rightarrow \mathtt{nat}, \mathtt{s} : \mathtt{nat} \times \mathtt{nat} \rightarrow \mathtt{nat} \rangle$*, then a* Σ*-algebra is the algebra of natural numbers: Namely,* `nat` *is interpreted as the set of natural numbers* ω*, the element* 0 *interprets constant* `z` *and sum function interprets* `s`*.*

Referring to Example 11.2.1, signature Σ "resembles" the functor

$$\mathcal{N}(X) = \mathbf{1} + X \times X,$$

where 1 is a singleton set. A $\mathcal{N}$-algebra is a pair (A, α) where A is a set and $\alpha : \mathcal{N}(A) \rightarrow A$ is a function that given a set A either returns the element in $\mathbf{1}$ or a "new" element built out of two elements in A.

More generally, if $\Sigma = \{\sigma_1, ..., \sigma_s\}$ is a signature such that each operation σ_i has arity n_i, we can associate a functor

$$\mathcal{F}_\Sigma(U) = U^{n_1} + ... + U^{n_s}$$

(where U^0 is a singleton set containing an element of U) such that a Σ-algebra with carrier A, can be represented by a function $\mathcal{F}_\Sigma(A) \rightarrow A$.

We have now all the ingredients for defining coalgebras and point out their duality with respect to algebras. We restrict our definition to coalgebras over endo-functors of **Set**.

Definition 11.2.1 ($\mathcal{F}$-coalgebra) *Let* $\mathcal{F}$ *be an endo-functor on the category* **Set**. *A* $\mathcal{F}$-coalgebra *consists of a pair* (A, α) *such that* $\alpha : A \rightarrow \mathcal{F}(A)$.

This definition makes clear also the duality between $\mathcal{F}$-algebras and $\mathcal{F}$-coalgebras. Indeed they are functions whose domain and codomain are "reversed", namely, are arrows between the same objects but with opposite directions. Different directions can be interpreted as "construction" and "observation". An $\mathcal{F}$-algebra with carrier set A is a function $\mathcal{F}(A) \rightarrow A$ and says how to "construct" elements of A by applying operations detailed by $\mathcal{F}$. On the other hand, a $\mathcal{F}$-coalgebra is a function $A \rightarrow \mathcal{F}(A)$ which, given an element of A, returns informations on the element. For instance let us consider $\mathcal{T}(X) = L \times X$, where L is a fixed set, then the coalgebra $\alpha : Q \rightarrow L \times Q$ can be though of as an automaton such that, for each state $q \in Q$, if $\alpha(q) = (l, q')$ then q' is the successor state of q reached with a transition labelled l.

[1] In the general case of multisorted signatures, also sorting must be preserved.

11.3 Transition Systems as Coalgebras

This section gives the preliminary definitions and notations on automata. We present a formal framework that, starting from ordinary automata, introduces the coalgebraic version of automata theory and the coalgebraic definition of HD-automata.

In the following we will use terms 'automaton' and 'transition system' interchangeably.

Definition 11.3.1 (Automata) *An* automaton A *is a triple* $(S, L, \rightarrow)$ *where* S *is the set of* states, L *the set of* actions *or* labels *and* $\rightarrow \subseteq S \times L \times S$ *is the* transition relation. *Usually, one writes* $s \xrightarrow{\ell} d$ *to indicate* $(s, \ell, d) \in \rightarrow$; s *is the* source *state and* d *is the* destination *or* target *state. Transition* $s \xrightarrow{\ell} d$ *is also called '*arrow*'.*

Observation 11.3.1 *In classical automata theory, an initial state* $\bar{s} \in S$ *is specified for automata. For the moment, we ignore initial states that will be specified when necessary.*

Depending on the transition relation, we can distinguish various classes of automata. For instance, *deterministic* automata are those automata having a transition relation which is functional, i.e. $s \xrightarrow{\ell} d$ and $s \xrightarrow{\ell} d'$ if, and only if, $d = d'$. Deterministic automata have one possible successor state for each state s and each label ℓ. *Non-deterministic automata* are automata which admit more than one possible successor for a state and a label.

We aim at developing a coalgebraic description of the minimization procedure, hence, we rephrase coalgebras in terms of structures that are more concrete than functors. Indeed, we provide a (concrete) representation of the terminal coalgebra (of an endofunctor over `Set`) in terms of sets and *quadruples* which will yield the minimal transition system. In particular, we define *bundles* as the concrete structures that are associated by the co-algebraic functor to states that will be (concretely) represented as objects of `Set`. In this way it is possible to express the functional aspect of the functor (at each step of the minimization algorithm) by means of particular structures that will be introduced in the following.

We report here definitions and notations taken from [73] that are hereafter used:

- $Q : \texttt{Set}$ denotes a set and $q : Q$ denotes an element in the set Q;
- `Fun` is the collection of functions among sets (the arrows of category **Set**). The function space over sets will have the following structure:

$$\texttt{Fun} = \{H \mid H = \langle S : \texttt{Set}\ , D : \texttt{Set}\ , h : S \rightarrow D \rangle\}.$$

  By convention we use S_H, D_H and h_H to respectively denote domain, codomain and mapping of an element of `Fun`.

A finite-state transition system can be coalgebraically described by employing two ingredients: A set Q, that represents the state space, together with a function $K : Q \to \wp_{\text{fin}}(L \times Q)$ that represents the "behaviour" of the transition system: $K(q)$ is the set of pairs (ℓ, q') such that $q \xrightarrow{\ell} q'$.

Definition 11.3.2 (Bundles) *Let L be the set of labels (ranged over by ℓ), then a* bundle *β over L is a structure $\langle D : \mathtt{Set}, Step : \wp_{fin}(L \times D)\rangle$. We call the first component of a bundle β* the support *of β. Given a fixed set of labels L, by convention, B^L denotes the collection of bundles and $\beta : B^L$ means that β is a bundle over L.*

Intuitively, the notion of bundle has to be understood as giving the data structure representing all the state transitions out of a given state. It details which states are reachable by performing certain actions.

Once a set of labels L has been fixed, we can consider the polynomial endofunctor $\mathcal{A}(X) = \wp_{\text{fin}}(L \times X)$ in $\mathtt{Set}$. Functor $\mathcal{A}$ operates on both sets and functions, and characterizes a whole category of labelled transition systems, i.e. of coalgebras. The following clauses define $\mathcal{A}$.

- $\mathcal{A}(Q) = \{\beta : B^L \mid D_\beta = Q\}$, for each $Q : \mathtt{Set}$;
- For each $H : \mathtt{Fun}$, $\mathcal{A}(H)$ is defined as follows:
  - $S_{\mathcal{A}(H)} = \mathcal{A}(S_H)$ and $D_{\mathcal{A}(H)} = \mathcal{A}(D_H)$;
  - $h_{\mathcal{A}(H)}(\beta : \mathcal{A}(S_H)) = \langle D_H, \{\langle \ell, h_H(q)\rangle \mid \langle \ell, q\rangle : Step_\beta\}\rangle$.

A labelled transition system over a set of labels L is a coalgebra for functor $\mathcal{A}$, namely it is a function K such that $D_K = \mathcal{A}(S_K)$. Note that the convention on functions permits us not to mention the carrier of a coalgebra K it is implicitly given by S_K.

We can rephrase the concepts of coalgebras homomorphism and finality for transition systems:

Definition 11.3.3 (Homomorphism and finality of transition systems) *Let K and F be two transition systems. A function H is a* homomorphism *of transition system if*

$$S_H = S_K, \qquad D_H = S_F, \qquad H; F = K; \mathcal{A}(H).$$

Which can be represented by the following commuting diagram:

$$\begin{array}{ccc} S_K & \xrightarrow{h_H} & S_F \\ {\scriptstyle K}\downarrow & & \downarrow{\scriptstyle F} \\ \mathcal{A}(S_K) & \xrightarrow[\mathcal{A}(h_H)]{} & \mathcal{A}(S_F) \end{array}$$

A transition system F is final *if, for any other transition system K, there is a unique homomorphism from K to F.*

As usual, homomorphisms correspond to the idea of functions which commutes with the coalgebraic operations while finality encompass the idea of minimality. Indeed, final transition systems are those transition systems that are image of all other transition systems in a certain class through functions which preserves their "behaviour". General results (e.g. [4]) ensure the existence of the final coalgebra for a large class of functors. These results apply to the functors defining transition systems. In particular, it is interesting to see the result of the iteration along the terminal sequence [175] of functor $\mathcal{A}$.

Let K be a transition system, and let $H_0, H_1, \ldots, H_{i+1}, \ldots$ be the sequence of functions computed by $H_{i+1} = K;\widehat{\mathcal{A}(H_i)}$, where H_0 is the unique function from S_K to the one-element set $\{*\}$ given by $S_{H_0} = S_K$; $D_{H_0} = \{*\}$; and $h_{H_0}(q : S_{H_0}) = *$. In [73], the following result is stated:

Theorem 11.3.1 *Let K be a finite-state transition system. Then,*

- *The iteration along the terminal sequence converges in a finite number of steps, i.e. $D_{H_{i+1}} \equiv D_{H_i}$,*
- *The isomorphism mapping $F : D_{H_i} \rightarrow D_{H_{i+1}}$ yields the minimal realization of transition system K.*

Chapter 12

Verification of History Dependent Automata

Abstract

Modeling systems with mobile processes is quite reasonable under many point of view. Process calculi have been usefully exploited both for theoretical investigation and for studying "concrete" aspects of mobile systems. Unfortunately, even if many proposed theoretical frameworks are quite adequate for considering most issues related to distributed and concurrent systems, they do not fit well with respect to verification purposes. HD-automata extends classical automata and have been proposed for explicitly modeling phenomena related to history dependent formalism at the operational level.

This chapter reviews the co-algebraic specification of HD-automata and a minimization algorithm based on the co-algebraic definition of HD-automata proposed in [135, 73].

Contents

12.1 History Dependent Automata

Verification of systems that can be adequately modeled as mobile processes is difficult because many "source of infinity" can be introduced. For instance, let us consider transition systems obtained from π-agents, we have that transitions can generate new names. Indeed, let us consider the $(OPEN)$ rule of π-calculus:

$$\frac{p \xrightarrow{\overline{x}y} q}{(\nu\ y)p \xrightarrow{\overline{x}(y)} q} \quad \text{if } x \neq y. \tag{12.1}$$

Such rule basically establishes that a state $((\nu\ y)p)$ can create a new name and can export it over a channel x. Notice that the state of the transition system corresponding to $((\nu\ y)p)$ represents a point of the computation where y "does not exists", while the target state of the bound output transition is a point of the computation where y "becomes available". Rule (12.1) introduces an infinite branching in the automata corresponding to agents that perform bound output transitions.

As already noticed in Section 3.1.2, the rule for input transition of the early semantics of the π-calculus also introduces infinite branching because it is necessary to consider a transition for *any* name that instantiates the input parameter.

Let us remark that it is of course reasonable (and desirable) to model π-calculus semantics with rules as (12.1) or by means of the early semantics because such rules account for scope extrusion of names that is one of the major peculiarities of π-calculus and permits to model and reason on many aspects of mobile systems. On the other hand, since those kind of semantics had been introduced without considering verification issues, such rules are problematic when verification purposes are under consideration.

A different phenomenon that produces infinite automata is due to name extrusion. A possible "implementation" of name extrusion is to reserve an infinite sequence of names from which a new name can be taken when a transition extrudes a fresh name. This approach has been proposed and analyzed in [153, 74]. A drawback of this approach is that an infinite number of states is generated in the case of agents with infinite behaviour. Indeed, let us consider the agent $A(x) = (\nu\ y)\bar{x}y.A(y)$. Agent $A(x)$ generates a new name y, emit it along x and continues as $A(y)$. This behavior is "encoded" in the approach of [153, 74] as

$$A(x) \xrightarrow{\bar{x}(x_0)} A(x_0) \xrightarrow{\bar{x_0}(x_1)} A(x_1) \xrightarrow{\bar{x_1}(x_2)} A(x_2)...$$

Hence, to obtain finite state automata also for agents with infinite behaviour, we need a mechanism to model "resource deallocation". Let us again consider agent A; after each bound output $\bar{x_i}(x_{i+1})$ transition, the name x_i will never be used in future transitions, hence we could re-use it provided that a mechanism for re-stating its freshness is given.

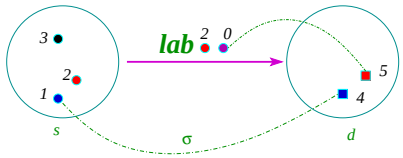


Figure 12.1: A HD-automaton transition

In order to model this kind of evolution in a framework suitable for verifying systems it is necessary to enrich the structure of states and transitions of ordinary transition systems.

History Dependent automata (HD-automata in brief) have been proposed in [151, 134, 135, 73] as a new operational model for history dependent calculi, namely those calculi whose semantics is defined in terms of a labelled transition system such that the labels may carry information generated in the past transitions of the system and this "historical" information can influence the future behaviour of the system. Probably the simplest history dependent calculus is CCS with value passing [104], another example is the CCS with locality [25]; finally, as we have seen π-calculus LTS semantics all have labels that can contain names generated in past transitions[1].

HD-automata aim at giving a finite representation of otherwise infinite label transition systems. Similarly to ordinary automata, HD-automata are made out of states and labelled transitions. Their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt as syntactic components of labels, but become an explicit part of the operational model. This permits to model name creation/deallocation or name extrusion that are typical linguistic mechanisms of name passing calculi.

An important aspect of HD-automata to emphasize is that names of a state have *local meaning*. For instance, if $A(x, y, z)$ denotes an agent having three free names x, y and z, then agent $A(y, x, z)$ is different from $A(x, y, z)$, however, they can be represented by means of a single state q in a HD-automaton simply by considering a "swapping" operation on the names (corresponding to) x and y of q. More generally, states that differs only for renaming of their local names are identified in the operational model.

Local meaning of names requires a mechanism for describing how names correspond each other along transitions. Graphically, we can represent such correspondences using "wires" that connect names of label, source and target states of transitions. For instance, Figure 12.1 depicts a transition from source state s to destination state d. The transition exposes two names: Name 2 of s and a fresh name 0. State s has three names, 1, 2 and 3 while d has two names 4 and 5 which correspond to name 1 of s and to the new name 0, respectively. Notice that names 3 is discharged along such transition.

[1]Also formalism that are not related to process calculi can be considered as history dependent; for instance, Petri nets [90] are a paradigmatic history dependent formalisms.

As described in Figure 12.1, HD-automata relies on the fact that names are local. This allows for a compact representation of agent behaviour by collapsing states that differ only for renaming of local names encompasses the main characteristics of name-passing calculi, namely, creation/deallocation of names. Indeed, name creation is simply handled by associating in the target state a name not in the source state.

A computation performed on a HD-automaton associates a "history" to names of the states appearing in the computation, in the sense that it is possible to reconstruct the associations which lead to the state containing the name. Clearly, if a state is reached in two different computations, different histories could be assigned to its names.

Various families of HD-automata have been introduced. Roughly speaking each class of HD-automata corresponds to a class of history dependent calculi or different behavioural semantics. The reader is referred to [151] for details. In the next sections we will present a coalgebraic definition of HD-automata for π-calculus for the early semantics. The coalgebraic presentation of HD-automata for π-agents has been introduced in [135]. In order to make this part of the dissertation self-contained, we report (with slight variations on the notation) the presentation appeared in [73] of the minimization algorithm for HD-automata.

12.2 HD-automata for π-agents

This section borrows the minimal notations and definitions introduced in [73] that will be used in Chapter 13.

Names appear explicitly in the states of an HD-automaton: The idea is that the names associated to a state play a rôle in the state evolution. Let $\mathcal{N}$ be an infinite countable set of names ranged over by v and let $\mathcal{N}^\star$ be the set $\mathcal{N} \cup *$, where $* \notin \mathcal{N}$ is a distinguished name and will be used for modeling name creation. We also assume that $<$ is a total order on $\mathcal{N}^\star$ (for instance, it can be the lexicographic order on $\mathcal{N}$ and $\forall v \in \mathcal{N} : * < v$). Given a state q of a HD-automaton, a set $\{v_1, \ldots, v_{|q|}\} \subseteq \mathcal{N}$ of local names and a permutation group G_q are associated with q. Elements of G_q are those permutations of names of q that leave unchanged the behaviour of q. Moreover, the identity of names is local to the state: States which differ only for the order of their names are identified. Due to the usage of local names, whenever a transition is performed a name correspondence between the name of the source state and the names of the target state is explicitly required.

Table 12.1 is a synoptic collection of definitions from [73] (with few notational changes); it reports the definitions of *named sets*, *named functions*, and composition of named functions. In Table 12.1 and in the following, the general product $\prod$ is employed (as usual in type theory) to type functions f such that the type of $f(q)$ is dependent on q.

A named set represents a set of states equipped with a mechanism to give local meaning to names occurring in each state. In particular, function $|\_|$ yields the

Named set A *named set* A is a structure

$$A = \langle Q : \mathtt{Set}, |\_| : Q \longrightarrow \omega, \leq : \wp(Q \times Q), G : \prod_{q \in Q} \wp(\{v_1..v_{|q|}\} \xrightarrow{bij} \{v_1..v_{|q|}\}) \rangle$$

where $\forall q : Q_A$, $G_A(q)$ is a permutation group and $\leq_A$ is a total ordering.

Named function A *named function* H is a structure

$$H = \langle S : \mathtt{NSet}, D : \mathtt{NSet}, h : Q_S \longrightarrow Q_D, \Sigma : Q_S \longrightarrow \wp(\{h(q)\}_D \xrightarrow{inj} \{q\}_S \cup *) \rangle$$

where $\forall q : Q_{S_H}, \forall \sigma : \Sigma_H(q)$,

1. $G_{D_H}(h_H(q)); \sigma = \Sigma_H(q)$ and
2. $\sigma; G_{S_H}(q) \subseteq \Sigma_H(q)$.

Composition of named functions Named functions can be composed in the obvious way. Let H and K be named functions. Then $H; K$ is defined only if $D_H = S_K$, and

$$S_{H;K} = S_H, \qquad D_{H;K} = D_K, \qquad h_{H;K} : Q_{S_H} \longrightarrow Q_{D_K} = h_H; h_K,$$
$$\Sigma_{H;K}(q : Q_{S_H}) = \Sigma_K(h_H(q)); \Sigma_H(q)$$

Let H be a named function, $\widehat{H}$ denotes the surjective component of H:

- $S_{\widehat{H}} = S_H$ and $Q_{D_{\widehat{H}}} = \{q' : Q_{D_H} \mid \exists q : Q_{S_H}.h_H(q) = q'\}$,
- $|q|_{D_{\widehat{H}}} = |q|_{D_H}$, $G_{D_{\widehat{H}}}(q) = G_{D_H}(q)$, $h_{\widehat{H}}(q) = h_H(q)$ and $\Sigma_{\widehat{H}}(q) = \Sigma_H(q)$

Table 12.1: Definitions

number of local names of states. Moreover, the permutation group $G_A(q)$ allows one to describe directly the renamings that do not affect the behaviour of q, i.e., symmetries on the local names of q. Finally, we assume that states are totally ordered. By convention we write $\{q : Q_A\}_A$ to indicate the set $\{v_1, ..., v_{|q|_A}\}$ and we use `NSet` to denote the universe of named sets.

As in the case of standard transition systems, name functions are used to determine the possible transitions of a given state. Intuitively, definition of named function in Table 12.1 says that, for each state q in S_H, $h_H(q)$ yields the behaviour of q, i.e. the transitions departing from q. Since states are equipped with local names, a name correspondence (the mapping H_h) is needed to describe how names in the destination state are mapped into names of the source state, therefore we must equip H with a *set* $\Sigma_H(q)$ of injective functions. For a discussion on the rôle of symmetries with respect named function we refer the reader to [73].

12.2.1 Bundles over π-calculus actions

We want to represent the transition system for the early semantics of π-calculus [130]. The notion of bundle must be enriched.

First we have to fix the set of labels of transitions. Labels of transitions must distinguish among the different meanings of names occurring in π-calculus actions, namely synchronization, bound/free output and bound/free input. The set of π-calculus labels L_π is the set $\{TAU, BOUT, OUT, BIN, IN\}$. We specify two different labels for input actions: Label BIN is used when the input transition acquires a new name, namely a name that was not previously known to the agent, while IN corresponds to an input transition that acquires an already known name.

Since names are local to states, it is necessary to specify how label names are related to names of states. For instance, no name is associated to synchronization labels, whereas one name, is associated to bound output labels. Let $|\_|$ be the weight map associating to each π-label the set of indexes of distinct names the label refers to. The weight map is defined as follows:

$$|TAU| = \emptyset \quad |BOUT| = |BIN| = \{1\} \quad |OUT| = |IN| = \{1, 2\}$$

Table 12.2 collects definitions of *bundles* and *names of bundles* from [73].

A bundle on π-labels is defined as in Table 12.2. As above, the intuition is that the *Step* component of a bundle describes the set of successor states for a given source state. More precisely, if $\langle \ell, \pi, \sigma, q \rangle \in qd\ \mathcal{D}$, then q is the destination state; ℓ is the label of the transition; π associates to the label the names observed in the transition; and σ states how names in the destination state are related with the names in the source state. According to the definition of σ in Table 12.2, a name in a destination state of a quadruple is mapped on the distinguished name $*$ only on transitions where a new name is created (i.e. transitions labelled with $BOUT$ or BIN).

Bundles A *bundle* β consists of the structure

$$\beta = \langle\, \mathcal{D} : \mathtt{NSet}, Step : \wp(qd\,\mathcal{D}) \,\rangle$$

where $qd\,\mathcal{D}$ is the set of *quadruples* of the form $\langle \ell, \pi, \sigma, q\rangle$ given by

$$qd\,\mathcal{D} = \{\langle \ell : L_\pi, \pi : |\ell| \xrightarrow{inj} \{v_1..\}, \sigma : \prod_{\ell \in L_\pi} \{q\}_{\mathcal{D}} \xrightarrow{inj} Q\ell, q : Q_{\mathcal{D}}\rangle\}.$$

and

$$Q\ell = \begin{cases} \mathcal{N}^\star & \text{if } \ell \in \{BOUT, BIN\} \\ \mathcal{N} & \text{if } \ell \notin \{BOUT, BIN\} \end{cases}$$

under the constraint that $G_{\mathcal{D}_\beta}(q); S_q = S_q$, where $S_q = \{\langle \ell, \pi, \sigma, q\rangle \in Step_\beta\}$ and $\rho; \langle \ell, \pi, \sigma, q\rangle = \langle \ell, \pi, \rho; \sigma, q\rangle$.

Bundle names Let β be a bundle. Function $\{\!|\, \_ \,|\!\} : B \to \mathcal{N}$, mapping each bundle to the set of its names, is defined by

$$\{\!|\, \beta \,|\!\} = \bigcup_{\langle \ell, \pi, \sigma, q\rangle \in Step_\beta} rng(\pi) \cup rng(\sigma) \setminus \{*\}$$

where rng yields the range of functions. We only consider bundles β such that $\{\!|\, \beta \,|\!\}$ is finite and we let $\lfloor \beta \rfloor$ to indicate the number of names which occur in the bundle β (i.e. $\lfloor \beta \rfloor = |\{\!|\, \beta \,|\!\}|$).

Table 12.2: Definitions

In order to exploit named functions for representing HD-automata it is necessary to equip the set of bundles B with a named set structure. In other words we must define a total order on bundles, a function that maps a bundle to its number of names and a group of permutations over those names. Table 12.2 reports the definition of names of a bundle which are the names (different from $*$) that appear either in the labels or in the range of σ's of the quadruples of the bundle.

The minimization algorithm necessitates of a mechanism for determining the representative element of a given class of equivalent states. Intuitively, two states are equivalent when they have the "same" bundles, hence, the choice of a canonical state turns in the choice of a canonical bundle. We can assume that a total order on states and labels exist. Hence, quadruples are totally ordered, e.g. assuming the lexicographic order of labels, states and names. The order over quadruples yields an ordering $\sqsubseteq$ over bundles. This ordering relation will be used to define canonical representatives of bundles. The ordering on quadruples can be non ambiguously defined only assuming an ordering on the support of bundles.

Finally, the group of a bundle can be defined once we define how a permutation is applied to a bundle. Given a bundle β and a permutation $\theta : \{\!|\, \beta \,|\!\} \xrightarrow{bij} \{\!|\, \beta \,|\!\}$, bundle $\beta;\theta$ is defined as $\mathcal{D}_{\beta;\theta} = \mathcal{D}_\beta$, $step_{\beta;\theta} = \{\langle \ell, \pi; \theta, \sigma; \theta, q\rangle \mid \langle \ell, \pi, \sigma, q\rangle : \beta\}$; the group of β ($Gr\ \beta$) is defined as $Gr\ \beta = \{\rho | \beta; \rho^\star = \beta\}$.

12.2.2 Normalizing bundles

Normalization is the most important operation on bundles. It is necessary because (i) we must establish a canaonical way of chosing the step component of a bundle among a number of different equivalent ways; (ii) more importantly, *redundant* input transitions must be removed. Indeed, redundant transitions occur when an HD-automaton is built from a π-calculus agent. During this phase, it is not possible to decide which free input transitions are required, and which transitions are covered by the bound input transition[2]. The solution to this problem consists of adding a superset of the required free input transitions when the HD-automaton is built, and to exploit a reduction function to remove the ones that are unnecessary.

The *normalization* function $norm(\beta)$ is defined as follows:

- $\mathcal{D}_{norm(\beta)} = \mathcal{D}_\beta$
- $Step_{norm(\beta)} = min_{\sqsubseteq}\big(Step_\beta \setminus \{\langle IN, xy, \sigma, q\rangle \mid y \notin an_\beta\}\big)$,

where the auxiliary definitions (taken from [73]) are reported in Table 12.3.

The order relation $\sqsubseteq$ is used to define the canonical representatives of bundles and relies on the order of quadruples. In the following, we use $perm(\beta)$ to denote the canonical permutation that associates $Step_{norm(\beta)}$ and $Step_\beta \setminus \{\langle IN, xy, \sigma, q\rangle \mid y \notin an_\beta\}$.

[2]In the general case, to decide whether a free input transition is required it is as difficult as to decide the bisimilarity of two π-calculus agents.

red Reduction function $red(\beta)$ on bundles is defined as follows: • $\mathcal{D}_{red(\beta)} = \mathcal{D}_\beta$, • $Step_{red(\beta)} = Step_\beta \setminus \{\langle IN, xy, \sigma, q\rangle \mid \langle BIN, x, \sigma', q\rangle : Step_\beta \wedge \sigma' = \sigma; \{y \to *\}\}$. where $\sigma; \{y \to *\}$ is the function equal to σ on any name different from y and that assigns $*$ to y.
Active names The set of *active names of a bundle* β is $an_\beta = \{\!\lceil red(\beta) \rceil\!\}$.
Minimal step $min_\sqsubseteq$ is the function that, when applied to $Step_\beta$, returns the step of the minimal bundle (with respect to order $\sqsubseteq$) among those obtained by permuting names of β in all possible ways.

Table 12.3: Auxiliary definitions for *norm*

A canonical way of extracting $Gr\ \beta$, the group of permutations bundle β, is given by defining $Gr\ \beta = \{\rho \mid Step_\beta; (\rho[^*/_*]) = Step_\beta\}$.

12.2.3 The minimization algorithm

HD-automata for π-agents are particular transition systems over named sets defined as follows:

- the elements of the state Q_A are π-agents $p(v_1, ..., v_n)$ ordered lexicographically: $p_1 \leq_A p_2$ iff $p_1 \leq_{lex} p_2$
- $|p(v_1, ..., v_n)|_A = n$,
- $G_A q = \{id : \{q\}_A \longrightarrow \{q\}_A\}$, where id denotes the identity function,
- $h : Q_A \longrightarrow \{\beta \mid \mathcal{D}_\beta = A\}$ is such that $\langle \ell, \pi, \sigma, q'\rangle \in Step_{h(q)}$ represent the π-calculus transitions from agent q.

The bundle $Step_{h(q)}$ contains only a subset of representative transitions (represented as $q \xrightarrow{\ell,\pi,\sigma} q'$) from q as discussed in [73].

A transition system over L_π is a HD-automaton and can be co-algebraically specified as a named function K such that $D_K = \mathcal{T}(S_K)$. The action of functor $\mathcal{T}$ over named sets is given by:

- $Q_{\mathcal{T}(A)} = \{\beta : Bundle \mid \mathcal{D}_\beta = A, \beta \text{ normalized}\}$,

- $|\beta|_{\mathcal{T}(A)} = \lfloor \beta \rfloor$,
- $G_{\mathcal{T}(A)}(\beta) = Gr\ \beta$,
- $\beta_1 \leq_{\mathcal{T}(A)} \beta_2$ iff $Step_{\beta_1} \sqsubseteq Step_{\beta_2}$,

while the action of functor $\mathcal{T}$ over named functions is given by:

- $S_{\mathcal{T}(H)} = \mathcal{T}(S_H)$, $D_{\mathcal{T}(H)} = \mathcal{T}(D_H)$,
- $h_{\mathcal{T}(H)}(\beta : Q_{\mathcal{T}(S_H)}) : Q_{\mathcal{T}(D_H)} = norm(\beta')$,
- $\Sigma_{\mathcal{T}(H)}(\beta : Q_{\mathcal{T}(S_H)}) = Gr(norm(\beta')); (perm(\beta'))^{-1}; inj : \{\!|\, norm(\beta') \,|\!\} \longrightarrow \{\beta\}_{\mathcal{T}(S_H)}$ where $\beta' = \langle D_H, \{\langle \ell, \pi, \sigma'; \sigma, h_H(q) \rangle \mid \langle \ell, \pi, \sigma, q \rangle : Step_\beta, \sigma' : \Sigma_H(q)\} \rangle$.

A detailed discussion on the definition of functor $\mathcal{T}$ can be found in [73]. Function K is defined as

- $S_K = A$,
- $h_K(q) = norm(h(q))$,
- $\Sigma_K(q) = Gr(h_K(q)); (perm(h(q)))^{-1}; inj : \{\!|\, h(q) \,|\!\} \longrightarrow \{q\}_A$

The minimal HD-automata is built by an iterative procedure on K whose initial approximation is

- $S_{H_0} = S_K$, $D_{H_0} = unit$ where $Q_{unit} = \{*\}$, $|*|_{unit} = 0$ (and hence $\{*\} = \phi$), $G_{unit}\ * = \phi$, and $* \leq_{unit} *$,
- $h_{H_0}(q : Q_{s_{H_0}}) = *$,
- $\Sigma_{H_0} q = \{\phi\}$

and the iterative step is given by

$$H_{i+1} = \widehat{K; \mathcal{T}(H_i)}.$$

Quoting [73]:

Theorem 12.2.1 *Let K be a finite state HD-automaton. Then*

- *The iteration along the terminal sequence converges in a finite number of steps: n exists such that $D_{H_{i+1}} \equiv D_{H_i}$,*
- *The isomorphism mapping $F : D_{H_i} \longrightarrow D_{H_{i+1}}$ yields the minimal realization of the transition system K up to strong early bisimilarity.*

The following functional expression (in an extended λ-calculus) makes the iteration step of the normalization algorithm explicit.

$$h_{H_{i+1}} = (\lambda q.norm\ \langle A, \{\langle \ell, \pi, \sigma, q' \rangle \mid q \xrightarrow{\ell,\pi,\sigma} q'\}\rangle);$$
$$\lambda\beta.norm\ \langle D_{H_i}, \{\langle \ell, \pi, \sigma'; \sigma, h_{H_i}(q)\rangle \mid \langle \ell, \pi, \sigma, q\rangle : Step_\beta, \sigma' : \Sigma_{H_i}(q)\}\rangle$$

$$h_{H_{i+1}}(q) = norm\ \langle D_{H_i}, \{\langle \ell, \pi, \sigma'; \sigma, h_{H_i}(q')\rangle \mid q \xrightarrow{\ell,\pi,\sigma} q', \sigma' : \Sigma_{H_i}(q')\}\rangle.$$

Notice that the normalization on the transition system is absorbed by the normalization on the resulting bundle.

Chapter 13

Mihda: A Verification Environment

Abstract

This chapter presents Mihda, a verification environment centered around the minimization algorithm presented in Section 12.2. Mihda is written in ocaml [45] and the most relevant implementation choices are discussed also with respect to the ocaml features. We discuss the architectural aspects of the environment in Section 13.1, while the principal data structures are detailed in Section 13.2. Section 13.3.1 contains the main result of the chapter which is the correctness of the implementation of the minimization algorithm described in 12.2. In all these sections the emphasis is on tight connection between the formal co-algebraic specification of the framework and the ocaml implementation which nicely represents the interplay between theory and practice. Indeed, Mihda can be seen as the experimental validation of the theoretical framework proposed in [135, 73], ocaml features interestingly permit to prove correctness with respect to the theoretical framework and the experimental environment also suggests new issues for theoretical investigation.

Mihda can be downloaded from `http://jordie.di.unipi.it:8080/mihda`, where also an interactive interface (detailed in Chapter 14) is available.

Contents

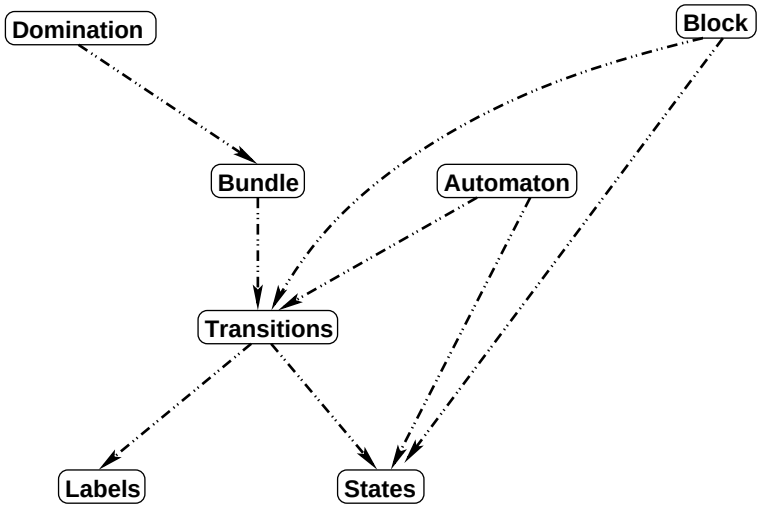


Figure 13.1: Mihda Architecture

13.1 Architectural Aspects of Mihda

The main features of ocaml exploited in our realization are polymorphism and encapsulation. Polymorphism is one of the intrinsic peculiarity of ML-language family, while encapsulation may be obtained in ocaml in two different ways; the first way is by using the object oriented features of the language, the second way is provided by modular programming features. More precisely, the module system separates the definition of interface specification, called *signatures* (i.e. definition of *abstract data types*) from their realizations, called *structures*. A structure may be parameterized using *functors*. An ocaml functor constructs new modules by mapping modules of a given signature on structures of other signatures.

Observation 13.1.1 *Object oriented programming simply adds to polymorphism and encapsulation (features already present in functional programming) hierarchical relations among abstract data types. However, in our case, those relations does not play any rôle and, therefore, have not been exploited.*

Language ocaml has been chosen also for other reasons. As detailed in Section 12.2, the algorithm has been specified in a "type-theoretic" style and the underlying type system makes use of parametric polymorphism. The type system of ocaml offers all the necessary features for handling those kind of type. As a further benefit, Mihda remains tightly close to its specification as we will prove later.

Figure 13.1 reports (part of) the modules of Mihda and represents the relationships and dependencies among those modules. Nodes of the graph represent both Mihda modules and their principal types, e.g. State is the module for states and contains a declaration State_t of its main type that declares the type of the states of the automata. Two modules in Figure 13.1 are connected when the upper module

declares a type to be the same as the lower module. For instance, the arc connecting Bundle and Transitions states that in module Bundle a type is declared such that State_t must be declare in State.

Mihda allows the user to specify automata type and, after having implemented some functionalities on such data structures, a general minimization algorithm is applied and the minimal realization of the automaton is returned. This choice gives the opportunity of applying the same algorithm to different kind of automata that can be defined, provided that they are specified in a manner that respects the constraints imposed by functors. Those constraints expresses equalities that types must satisfy as said before. Notice that this mechanism also aids in implementing different semantic minimization. Let us consider a scenario where, modules of Mihda have been specified for a given class of automata. Later, we decide that the equivalence relation that must be considered should be changed. Conceptually, the algorithm and type declarations for states and automata remain unchanged. Indeed, it would be reasonable to modify only the module Domination (and perhaps Transitions[1]) that is the Mihda module where the (type of) semantic relation is declared. Mihda architecture allows programmer in the scenario depicted above to write new code only for the module Domination.

An easy exercise that we have done is to exploit the architecture of Mihda to adapt minimization of HD-automata to minimization of ordinary automata (we refer the interested reader to the web page of the Mihda project for detailed comments).

13.2 Main data structures

We discuss here the main data structures used in Mihda together with their most important properties. Moreover, their relations with the "theoretical" objects they implement is pointed out.

In the following sections we will use typewriter symbols to denote names for ocaml functions and variables. A list `l` is written as $[\mathtt{e}_1; ...; \mathtt{e}_h]$ while $\mathtt{l}_i$ denotes its i-th element (i.e. $\mathtt{e}_i$). Finally, we write $\mathtt{e} \in \mathtt{l}$ to indicate that `e` is an item of list `l`.

As a general remark, we point out that finite sets will be generally represented as lists. We say that a list `x` corresponds to a finite set X if, and only if, for each element e in X there exists $\mathtt{e} \in \mathtt{x}$ such that `e` *corresponds* to e. Later, we will define various *correspondence* relations over elements which depend on the type they live in. Note that, for each element $e \in X$, many instances of `e` can occur in `x`. However, we will often apply the function `Utils.unique` which removes multiple occurrences of items in a list.

In the following we will exploit two auxiliary functions, `list_rem` and `list_diff`, that are reported below together with their main properties. We also state and prove some properties that will be used in the proof of the correctness of Mihda.

[1] In general, transition types depend on the semantic relation because the new relation might require information that must be added on the labels.

`list_rem el list` returns the list obtained from `list` by removing all the occurrences of items equal to `el`. Its definition is

```
let rec list_rem el = function
  | []     → []
  | e::es →
    if (compare e el) == 0
    then list_rem el es
    else e :: (list_rem el es)
```

Each element occurring in the list is not removed if different from `el`. Indeed, the following lemma holds:

Lemma 13.2.1 $\forall a.\forall b.\forall ls \neq b \wedge a \in ls \Rightarrow a \in (Utils.list\_rem\ b\ ls)$

PROOF. We proceed by induction on the length of `ls`. If `ls` = [] then the implication trivially holds because the antecedent is false. If `ls` = `e` :: `es` then, by definition,

```
if (compare e b) == 0
then list_rem b es
else e :: (list_rem b es)
```

If $e = b$, then the result of the function is `list_rem b es` and also $a \in es$ because $a \neq b$ and $a \in$ `ls`. Therefore, by applying the inductive hypothesis, we have the thesis. In $e \neq b$, then the result of `list_rem b ls` is `e` :: (`list_rem b es`) then the thesis holds because, either `a` = `e` or else `a` ∈ `es` and, by inductive hypothesis `a` ∈ (`list_rem b es`) which gives the proof. □

`list_diff l m` returns the list obtained by subtracting `m` from `l` and is defined as

```
let rec list_diff l = function
  | []     → l
  | e::es → list_diff (list_rem e l) es
```

Function `list_diff` enjoys the following property:

Lemma 13.2.2 *Let* `l` *and* `m` *be two lists. Then* $\forall el \in m.el \notin (list\_diff\ l\ m)$.

PROOF. We reason by induction on the length of `m`. If `m` = [] then the proposition trivially holds. Let `m` be `e`::`es`, then the result of the function is `list_diff` (`list_rem e l`) `es`. If `el` = `e` then, by Lemma 13.2.1, `el` ∉ (`list_rem el l`) and the thesis follows by the fact that the recursive call never adds anything to the result (avoiding the possibility of re-introducing `el`). On the other hand, if `el` ≠ `e` then `el` ∈ `es`, and the inductive hypothesis concludes the proof. □

13.2.1 HD-automata states, labels and transitions

A generic automaton (see Definition 11.3.1) is made of four ingredients: States, initial state, labels and arrows. As far as finite state automata are concerned, it is possible to represent automata by enumerating states and transitions.

Observation 13.2.1 *We assume that* $1, .., n$ *are the names of a state having* n *names. This assumption does not imply any loss of generality because names are local to states. We reserve* 0 *for denoting name* $*$*, the symbol used for denoting name creation in transitions (see page 206). A symmetry over* n *names may be simply expressed by means of a list of distinct integers, each belonging to segment* $1, ..., n$*; for instance if* ρ *is the permutation*

$$\begin{pmatrix} 1 & ... & n \\ i_1 & ... & i_n \end{pmatrix}$$

then the list $[i_1; ...; i_n]$ *is a representation in terms of list of integers of* ρ*. For instance,* $[2; 1; 3]$ *represents a permutation of* 3 *elements: Namely, the permutation that exchanges* 1 *and* 2*, and leaves* 3 *unchanged.*

In this case we say that $[i_1; ...; i_n]$ corresponds to ρ*.*

We adopt this conventions on names and permutations also for representing other functions on names. In particular, if $qd = \langle \ell, \pi, \sigma, q \rangle$ is a quadruple, π is represented by means of a list of integers `pi` whose length is $\mid \ell \mid$ and whose i-th position contains $\pi(i)$ (for $i = 1, ..., \mid \ell \mid$). Finally, σ is a list of integers `sigma` whose length is m, the number of names of q and whose i-th element is $\sigma(i)$ (for $i = 1, ..., m$). We say that `pi` (`sigma`) *corresponds to* π (σ).

As discussed in Section 12.1, HD-automata are an extension of ordinary automata in the sense that states and labels have a richer structure carrying information on names. A state may be concretely represented as a triple

type *State_t* =
| State **of** *id:* string $*$ *names:* int *list* $*$ *group:* (int *list*) *list*

Where *id* is the name of the state; *names* are the local names of the state and are represented as a list of integers; the third component of a state is its *group* which is the set of those permutations that leave the state unchanged. By the previous observation, we can represent it as a list of list of integers.

Definition 13.2.1 (States correspondence) *An element* State(`q`, `names`, `group`) corresponds *to a state* q *of a named set* $A = \langle Q, \mid \_ \mid, \leq, G \rangle$ *if, and only if,*

- $q \in Q$
- $\mid q \mid =$ `List.length names`

- `group` *corresponds to Group in terms of correspondence between lists and sets (i.e. reciprocal element-wise correspondence, as described in Section 13.2).*

We remark that, since, we concretely represent names as integers we exploit the integer order to induce an order to states; therefore, we do not explicitly mention it in Definition 13.2.1. Notice also that the first component of bundles is not represented. This is possible because a main design choice is that we always deals with bundles that are obtained by applying the iterative construction $H_{i+1} = \widehat{K;\mathcal{T}(H_i)}$. Therefore, the first component of these bundles always is S_K, the set of states of the initial automaton.

Arrows are represented as triples with a *source* state, a *label* and a destination state.

```
type labeltype = string * int list * int list

type Arrow_t = Arrow of
  source: State_t * label: labeltype * destination: State_t
```

Type of arrows relies on type for labels which are triples whose first components are the name of the action (for π-agents, a string among $TAU, BOUT, OUT, BIN, IN$); the second component of a label is the list of names exposed in the transition; finally, the last component of a label is a function mapping names in the destination to names of the source state. A simpler alternative definition could have been obtained by embedding the `labeltype` in Arrow. Although more adherent to the definition of bundle, this solution is less general than the one adopted, because different transition systems have different labels.

Now we can give the structure which represents automata:

```
type Automaton_t =
  start: State_t *
  states: State_t list *
  arrows: Arrow_t list
```

The first component is the initial state of the transition system, then the list of *states* and *arrows* are given.

Bundles rely on quadruples over named sets. Basically, a quadruple describes state transitions. Transitions are labelled and, our implementation represents part of information carried by quadruples into labels:

```
type quadtype = Qd of Arrow.labeltype * State_t

type Bundle_t = quadtype list
```

Type `quadtype` and Observation 13.2.1 allows us to state a precise connection between quadruples and objects that populate `quadtype`.

Definition 13.2.2 (Quadruple correspondence) *Given a quadruple* $qd = \langle \ell, \pi, \sigma, q \rangle$*, we say that* Qd((lab, pi, sigma), q) corresponds *to* qd*, if, and only if,*

- `lab` *is a string with value* ℓ*,*
- `pi` *corresponds to* π*,*
- `sigma` *corresponds to* σ*,*
- `q` *corresponds to* q*.*

Definition 13.2.3 (Automata correspondence) *Let* $K = \langle Q, \mathcal{T}(Q), k : Q \to \mathcal{T}(Q), S \rangle$ *be a named function representing an automaton for a* π*-agent. We say that* (q, qs, as, S) corresponds to K *iff,* qs *corresponds to* Q*; for each* $qd \in k(q)$ *there exists* $\mathtt{a} \in \mathtt{as}$ *such that, if* $\mathtt{a} = (\mathtt{s}, (\mathtt{lab}, \mathtt{pi}, \mathtt{sigma}), \mathtt{t})$*, then* Qd((lab, pi, sigma), t) *corresponds to* qd*, and, for each* $\sigma \in S(q)$ *there is* $\mathtt{s} \in \mathtt{S}$ *such that* S *corresponds to* S*.*

Previous definition allows us to easily compute $k(q)$ for each state q. Indeed, let us consider the function `Automaton.bundle` defined as

```
let bundle hda q =
  Bundle.from_arrow_list (List.filter
      (fun x → 0 = State.compare (Arrow.source x) s) (arrows hda))
```

Proposition 13.2.1 *If* `hda` *and* `s` *are an HD-automata and a state which respectively correspond to* k *and* s*, then* `bundle hda s` *corresponds to* $k(s)$*.*

Proof. All arrows in `hda` are first filtered in order to select those of them whose source state is `s`, then function `Bundle.from_arrow_list` transforms each of them in the quadruple obtained by discharging the source from the arrow. □

Our representation of bundles, symmetries and function of names allows a simple representation of operation on bundles in terms of list manipulation. For instance, let us consider the function

$$\{\!|\, \beta \,|\!\} = \bigcup_{\langle \ell, \pi, \sigma, q \rangle \in Step_\beta} rng(\pi) \cup rng(\sigma) \setminus \{*\}$$

which yield the names of a bundle β and is implemented by function `bundle_names`, reported below.

```
let names = function Qd((lab,pi,sigma),target) → (pi @ sigma)

let bundle_names bundle =
  Utils.unique (Utils.list_rem 0 (List.flatten (List.map names bundle)))
```

Function `bundle_names` applies `names` to each quadruple in the list `bundle` corresponding to β (`List.map names bundle`). This returns a list whose items are the names appearing in the quadruples of `bundle` that are obtained by merging the lists `pi` and `sigma`. Finally, all those lists of names are merged together (`List.flatten`), if present, 0 is removed (`Utils.list_rem`) and multiple occurrences are collapsed into a single occurrence (`Utils.unique`). It is easy to see that the following proposition hold:

Proposition 13.2.2 *If* `bundle` *corresponds to a bundle* β *then* $a \in \{\!|\, \beta \,|\!\}$ *if, and only if,* a *occurs in* `bundle_names bundle`*.*

Proof. By definition, $a \in \{\!|\, \beta \,|\!\}$ if, and only if, there exists a quadruple $qd = \langle \ell, pi, \sigma, q \rangle$ in $Step_\beta$ such that $a \in rng(\pi) \cup rng(\sigma) \backslash *$. Let `Qd((lab, pi, sigma), q)` be a quadruple in `bundle` which corresponds to β. Then, by definition `pi` $= [\pi(1); ...; \pi(|\, \ell \,|)]$ and `sigma` $= [\sigma(1); ...; \sigma(m)]$, where $m = card(rng(\sigma))$ then $a \in$ `pi@sigma`. By observing that `pi@sigma` $\in$ (`List.map names bundle`). Then, since the flattening operation on lists corresponds to set union, we have that

$a \in$ `Utils.unique (Utils.list_rem 0 (List.flatten (List.map names bundle)))`.

Finally, by hypothesis, $a \neq *$ and, therefore, by Lemma 13.2.1. □

As stated in Section 12.2.1, normalization is the most important operation on bundles. It needs *red* function to be computed. Function *red* is implemented as follows:

```
let red bundle =
  let dominated =
    List.filter
      (fun qd → None <> (Domination.dominated qd bundle))
      bundle in
    list_diff bundle dominated
```

Proposition 13.2.3 *If* `bundle` *corresponds to a bundle* β *then* `red bundle` *corresponds to* $red(\beta)$*.*

Proof. First, let us observe that `dominated` is the list of quadruples which corresponds to the set of input quadruples that are redundant. Indeed, `bundle` is filtered according to the function `Domination.dominated`, that returns `None` only if `qd` is not redundant. Finally, by Lemma 13.2.2, `Utils.list_diff` removes from those transitions from `bundle`. □

```
let normalize red bundle =
  let w_bundle = red bundle in
  let an = bundle_names w_bundle in
    rename (list_diff bundle (List.filter (remove_in an) bundle))
```

Proposition 13.2.4 *If* `bundle` *corresponds to a bundle* β *then* `normalize red bundle` *corresponds to* $norm(\beta)$.

PROOF. First, active names `an` are computed in order to filter `bundle` obtaining all redundant transition covered by some bound input transition. By Proposition 13.2.2 and Lemma 13.2.1 we can conclude that `an` corresponds to $\{\!|\,\beta\,|\!\}$. Lemma 13.2.2 and the fact that list filtering corresponds to test of set membership, ensure that from `bundle` all redundant transitions are removed. Indeed, `remove_in` is defined as

```
let remove_in an =
  function Qd((lab,pi,sigma),target) as qd →
    (lab = "in") && (not (List.mem (obj qd) an))
```

which computes exactly the redundancy condition. The last function applied to the so computed bundle is `rename` which shifts the local active names of a state with their position in the list of active names. Note that this is a safe operation because only the active names of a state are important and their meaning is local to the state, moreover, such renaming amount to compute the permutation of names that returns the normalized bundle as defined in Section 12.2.1. □

13.2.2 Block

The most important data structures are *blocks*. They represent action of the functor on states of the automata and contain all those information for computing the iteration steps of the algorithm expressed in a set theoretic framework. Blocks represent both (finite) named functions and partitions of an automaton (at each iteration of the algorithm). Hence, at the last iteration a block corresponds to a state of the minimal automaton. A block has the following structure:

```
type Block_t =
  Block of
    id     : string  *
    states : State_t list  *
    norm   : Bundle_t  *
    names  : int list  *
    group  : int list list  *
    Σ      : (State_t → (int * int) list list)  *
    Θ⁻¹    : (State_t → (int * int) list)
```

Field *id* is the name of the block and is used to identify the block in order to construct the minimal automaton at the end of the algorithm. Field *states* contains the states which are considered equivalent with respect the equivalence relation

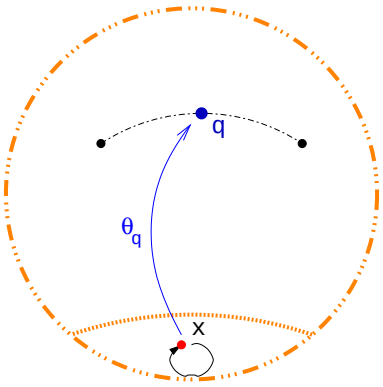


Figure 13.2: Graphical representation of a block

used in the algorithm[2]: In this case the early bisimulation relation. Remaining fields respectively represent

- the normalized bundle with respect to the block considered as state (*norm*),
- *names* is the list of names of the bundle in *norm*,
- *group* is its group,
- the functions relative to the bundle (Σ), last field, Θ^{-1}, is the function that, given a state q, maps the names appearing in *norm* into the name of q. Basically, $\Theta^{-1}(q)$ is the function which establishes a correspondence between the bundle of q and the bundle of the corresponding representative element in the equivalence class of the minimal automaton.

We draw (some components of) a block as in Figure 13.2: The upper elements are the states in the block, while the element `x` is the "representative state", namely it is a graphical representation of the block as a state. For each state q a function $\theta_{\mathtt{q}}$ maps names of `x` into the names of `q`. Function $\theta_{\mathtt{q}}$ describes "how" the block approximates the state `q` at a given iteration. The circled arrow on `x` aims at recording that a block also has symmetries on its names. Bundle *norm* of block `x` is computed by exploiting the ordering relations over names, labels and states.

A graphical representation of an iteration step of Mihda is given in Figure 13.3. The idea is that each block in the list of current blocks is first splitted, as far as possible, into a number of *buckets*, i.e. quasi-blocks defined later. Then each bucket is transformed in a new block, namely, the lacking components are uniformly computed at the end of the splitting phase.

The main operation on a block is the operation which splits a block into buckets. A list of blocks is returned as result of each iteration. Such blocks represent the states of the current approximation of the minimal automaton. A bucket has the same

[2]We recall that Mihda is parametrized with respect to the equivalence relation.

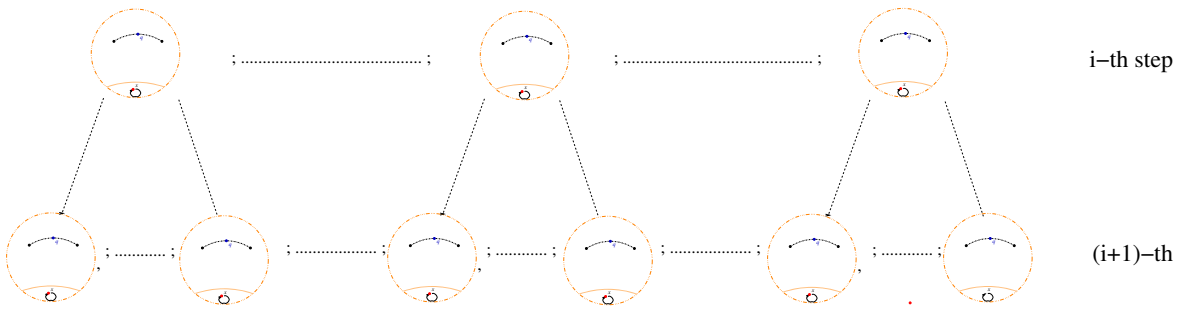


Figure 13.3: Graphical representation of an iteration step

fields of a block apart from the name, symmetries and the functions mapping names of destination states into names of source states. Basically, the split operation checks if two states in a block are equivalent or not. States which are no longer equivalent to the representative element of the block are removed and inserted into a bucket.

Given a bundle `bl`, a predicate over states `pred` and a block `block`, function `Block.split` returns a bucket and a block. The bucket collects the states of `block` which violate `pred`, while the returned block contains the remaining states.

```
let Block.split bl pred block =
  let eqv_chk = List.map (fun q → q, (pred q)) (states block) in
  let (eq_states, non_eq_states) =
    List.partition (fun (q,th) → th != None) eqv_chk in
  let new_states = (fst (List.split eq_states)) in
  let old_states = (fst (List.split non_eq_states)) in
  let new_inv_thetas = fun q →
    try
      invert ((function Some x→x) (List.assoc q eqv_chk))
    with Not_found → failwith "new_inv_thetas: exception" in
    (Bucket(new_states, bl, (Bundle.active_names bl), new_inv_thetas),
      (create
        (id block)
        old_states
        (norm block)
        (names block)
        (group block)
        (sigmas block)
        (inv_thetas block)))
```

It is worth to detail much more on the parameter `pred`. It is a function that, given a state `q`, returns an optional value that, roughly speaking, yields "the proof" for equivalence of `q` with respect to the other states of the bucket. More precisely, `pred` returns `None` if the equivalence does not hold, otherwise, a function θ mapping names of `q` into names of `bl` such that each arrow in the bundle of `q` appears in `bl`

(and *viceversa*). Function θ is the function Θ^{-1} associates to `q` when the bucket is turned into a block.

Note that, the new block has the same component of the old one because at the end of the splitting phase, all the states of the initial block will be assigned to a bucket without considering the information contained in those fields.

Definition below states the correspondence between a list of blocks and a coalgebra.

Definition 13.2.4 (Block correspondence) *Let $K = \langle Q, \mathcal{T}(Q), h : Q \to \mathcal{T}(Q)\rangle$ be a transition system over named sets and π-actions. A list of blocks* `blocks` *corresponds to K when given $q, q' \in Q$ and their* OCaml *representation* `q` *and* `q'`*, then $h(q) = h(q')$ if, and only if*

- *there exists* `bl` $\in$ `blocks` *such that* `q` *and* `q'` *are in* `bl`

and

- `bl.`$\underline{norm}$ *corresponds to* $Step_{h(q)}$

13.3 The main cycle

Let us recall the iterative step introduced at the end of Section 12.2.3:

$$h_{H_{i+1}}(q) = norm\ \langle D_{H_i}, \{\langle \ell, \pi, \sigma'; \sigma, h_{H_i}(q')\rangle \mid q \xrightarrow{\ell,\pi,\sigma} q', \sigma' : \Sigma_{H_i}(q')\}\rangle. \tag{13.1}$$

For each state q of the automaton, $h_{H_{i+1}}(q)$ determines the normalized bundle associated with to q. Following equation (13.1), we can compute $h_{H_{i+1}}$ over a finite state automaton in the following steps:

a. determine the bundle of q in the automaton;

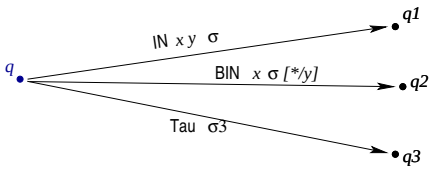
b. for each quadruple $\langle \ell, \pi, \sigma, q'\rangle$ in this bundle, apply h_{H_i} to q', the target state of the quadruple (yielding the bundle associated in the previous step to q');

c. left-compose this $\sigma' \in \Sigma(q')$ with σ;

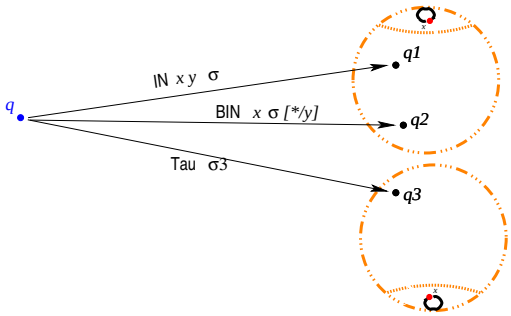
d. normalize the resulting bundle.

This intuitive idea must be refined because Mihda represents h_{H_i} as a list of blocks. In this representation, $h_{H_i}(q)$ corresponds to field $\underline{norm}$, namely the bundle of the block containing `q`, the state corresponding to q. A graphical representation of those steps in terms of blocks is depicted in Figure 13.4.

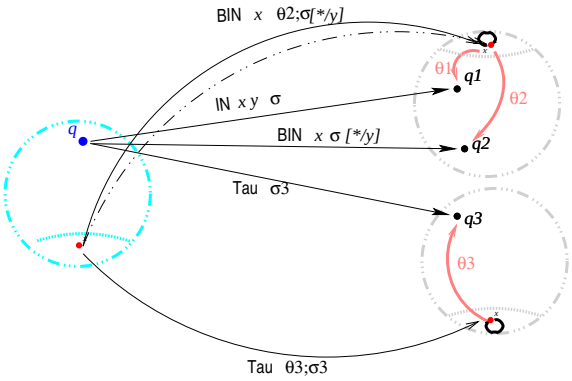
Step (a) is computed by the facility `Automaton.bundle` that filters all arrows of the automaton whose source corresponds to q. Figure 13.4(a) shows that a state `q` is taken from a block and its bundle is computed.



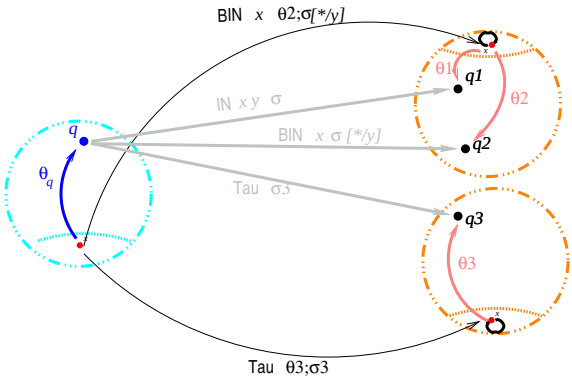
(a) Step 1



(b) Step 2



(c) Step 3



(d) Step 4

Figure 13.4: Computing $h_{H_{i+1}}$

Step (b) is obtained by applying `Block.next` to the bundle of `q`. `Block.next` substitutes all target states of the quadruples with the corresponding current block and computes the new mappings as described in Figure 13.4(b).
Step (c) seems not correctly adhere to the corresponding step of equation 13.1, but if we consider that θ functions are computed at each step by composing σ's we can see that they exactly play the rôle of σ's.
Finally, step (d) is represented in Figure 13.4(d) and is obtained via the function `Bundle.normalize`. Observe that redundant transitions must be removed and other components of the new block must be computed as defined by $\underline{\Theta^{-1}}$.

In order to give an intuitive understanding of the split operation, we describe how states are separated. Let us assume an automaton and an equivalence relations over states (e.g. early bisimilarity) have been fixed. Let `pred` be a function such that, given a bundle `bundle` and a state `q`, returns `None` if the normalized bundle of `q` in the automaton is not equivalent to `bundle` according to the equivalence relation between states `pred`. Then we can divide the states of a given block with respect to `bundle` and `pred` into two different lists of states. More precisely, this operation relies on the `Block.split` function and returns a pair whose first component is a bucket and whose second component is a block. The bucket contains those states whose normalized bundle is not equivalent to `bundle`, while the block component contains the remaining states.

The main part of Mihda consists of the cycle that computes the partitions of each iteration. Each block is splitted by iterating the application of function `split`.

```
let split blocks block =
  try
    let minimal =
      (Bundle.minimize red
        (Block.next
        (h_n blocks)
        (state_of blocks)
        (Automaton.bundle aut (List.hd (Block.states block))))) in
      Some (Block.split
        minimal
        (fun q →
          let normal =
            (Bundle.normalize
              red
              (Block.next (h_n blocks)
                (state_of blocks)
                (Automaton.bundle aut q))) in
            Bisimulation.bisimilar minimal normal)
        block)
  with Failure e → None
```

Let `block` be a block in the list `blocks`, function `split` computes `minimal` by minimizing the reduced bundle of the first state of `block`, and returns the optional value `Some`(`bk`, `block'`) if `Block.split` applied to `minimal`, to the bisimilarity relation and to `block` returns (`bk`, `block'`); otherwise `None` is returned.

Observation 13.3.1 *The choice of the state for computing* `minimal` *is not important: Without loss of generality, in fact, given two equivalent states* `q` *and* `q'`*, it is possible to map names of* `q` *into names of* `q'` *preserving their associated normalized bundle if, and only if, a similar map from names of* `q'` *into names of* `q` *exists.*

Moreover, we also point out that, minimization of a bundle with n names corresponds to normalize the bundle and replace each name with a name in $1, ..., n$ *preserving the convention that names of a state are an initial segment of natural numbers.*

Once `minimal` has been computed, `split` invokes `Block.split` with parameters `minimal`, `block`; the second argument of `Block.split` is a function that computes the (current) normalized bundle of each state in `block` and checks whether or not it is bisimilar to `minimal`.

This computation is performed by function `Bisimulation.bisimilar`. If bisimilarity holds through $\theta_{\mathtt{q}}$ then `Some` $\theta_{\mathtt{q}}$ is returned, otherwise `None` is returned.

We are now ready to comment on the main cycle of Mihda.

```
let blocks = ref [ (Block.from_states states) ] in
let stop = ref false in

  while not ( !stop ) do
    begin
      let oldblocks = !blocks in
      let buckets = split_iter (split oldblocks) oldblocks in
      begin
        blocks := (List.map (Block.close_block (h_n oldblocks)) buckets);
        stop :=
          (List.length !blocks) = (List.length oldblocks) &&
          (List.for_all2
            (fun x y → (Block.compare x y) == 0)
            !blocks
            oldblocks)
      end
    end
  done ;
  !blocks
```

Let `k` = (`start`, `states`, `arrows`) be a HD-automaton which corresponds to the coalgebra K. Initially, `blocks` is the list whose only item is a block containing all the states of `k`.

Observation 13.3.2 *By definition, initially,* `k` *corresponds to* H_0*, indeed, function* `Blocks.from_states` *puts all the states in the same block and assign the empty list to the field* norm *of such block.*

At each iteration, the list of blocks is splitted, as much as possible by `split_iter` that returns the list of buckets. Then, by means of `Block.close_block`, all buckets are turned into blocks which are assigned to `blocks`. Finally, the termination condition `stop` is evaluated. Note that Theorem 12.2.1 states that D_{i+1} must be isomorphic to D_i. This condition is equivalent to say that a bijection can be established between `oldblocks` (that corresponds to D_i) and `blocks` (corresponding to D_{i+1}). Moreover, since order of states, names and bundles is always maintained along iterations, both lists of blocks are ordered. Hence, the condition reduces to test whether `blocks` and `oldblocks` have the same length and that blocks at corresponding positions are equal. More formally, the following theorem holds:

Theorem 13.3.1 *For each iteration* i*, at the end of the main cycle of* Mihda*,* `blocks` *corresponds to* h_{H_i}*.*

Proof. The proof is given by induction on the iteration step. The base of induction is trivial.

Let assume that, at the end of the i-th iteration, the theorem holds and, by contradiction, that, at the end of the $(i+1)$-th iteration `blocks` does not correspond to $h_{H_{i+1}}$. Then, by Definition 13.2.4, there are two states q and q' such that either

1. `q` and `q'` lies in different blocks

or else

2. both `q` and `q'` are in the same block `bl` but `bl.`*norm* does not corresponds to $Step_{h_{K_i}(q)}$.

Let us first consider Case (1). By hypothesis, $h_{K_{i+1}}(q) = h_{K_{i+1}}(q')$ then $h_{K_i}(q) = h_{K_i}(q')$ because it is not possible that states distinguished in a given iteration later become bisimilar. By inductive hypothesis, `blocks` at i-th iteration corresponds to h_{K_i}, hence `q` and `q'` are in the same block at the i-th iteration. States `q` and `q'` can be separated at the $(i+1)$-th iteration, if, by construction, `split` assigns them to different buckets. This can happen only if there exists a state whose minimized bundle computed in `minimal` can be put in correspondence by `Bisimulation.bisimilar` with the normalized bundle of `q` but not with the normalized bundle of `q'`. Since `minimize` simply renames bundles preserving the order of names, then, at the $(i+1)$-th iteration, the normalized bundle of `q` corresponds to $h_{K_{i+1}}(q) = h_{K_{i+1}}(q')$ but the normalized bundle of `q'` does not correspond to $h_{K_{i+1}}(q')$. This contradicts Proposition 13.2.4 that ensures that `normalize` correctly implements function *norm*.
Following the final part of the proof of Case (1), we can simply observe that

Case (2) is not possible by construction. Indeed, the field `Block.`*norm* is built out from the normalized bundle of its states that, by Proposition 13.2.4, it must correspond to normalized bundles of the corresponding states of the automaton. □

Theorems 13.3.1 and 12.2.1 also ensure that the termination condition is correctly computed because, by Theorem 12.2.1, a fix-point is reached and, at the terminal iteration `blocks` corresponds to the fix-point (Theorem 13.3.1). The successive iterations will not change any block therefore, the length of `blocks` and each block in it will not change. As stated above, `blocks` and its elements are maintained ordered along each iteration and therefore, checking whether `blocks` change or not can be executed, as expressed by the assignment to `stop` in the main cycle, checking if $\texttt{blocks}_i$ equals $\texttt{oldblocks}_i$ for any block $\texttt{blocks}_i \in \texttt{blocks}$.

13.4 Concluding Remarks

The choice of implementing Mihda in ocaml has been driven by the functional and type-theoretic flavour of the co-algebraic specification. Both features fit well with ocaml programming characteristics. At a first glance, one can think that performance is undertaken: Surprisingly our benchmarks suggest that Mihda is not inefficient. For instance, we have considered the specification of the core of the handover protocol for the GSM Public Land Mobile Network proposed by the European Telecommunication Standards Institute [143]. The HD-automaton obtained by the π-calculus specification has 506 states and 745 transitions. The minimization of the handover protocol takes almost 9 seconds on an Athlon 1800+ under Linux RedHat 7.2, while 21 seconds are necessary on a Pentium III 500Mhz under Linux RedHat 7.1. The resulting HD-automaton consists of 105 states and 197 transitions.

The phase where Mihda spends the most part of computation time is the calculation of symmetries of names. For the time being Mihda trivially computes symmetries by generating the permutations and checking whether or not they change the behaviour of blocks. In this way the computational cost is factorial in the number of names. This implementation choice was suggested by the goal of producing a prototype of Mihda rapidly. However, the number of names remains very low in real cases. For instance, the specification of the GSM protocol initially consists of π-agents with 11 names. Instead, the average number of names per state in the compiled HD-automaton is 2.4: only two states have five names and more that three hundred states have only two names. We plan to enhance the efficiency of symmetries computation. Indeed, by considering that symmetries on new names added to a block does not affect already computed symmetries, we can compute new symmetries and "multiply" them with the old ones.

Chapter 14

Verification on the Web

Abstract

This chapter defines a web interface for Mihda. In the following we discuss how verification environment can be turned into *web services* and integrated with simple programming mechanism. As a case study, we show how Mihda and a different verification environment, HAL, can be used in a unique framework. Despite of its simplicity, the approach also allows to get rid of platform heterogeneity.

The framework also permits to exploits the facilities of both the environments as they were "programming libraries" for constructing verification applications "on the web".

Contents

14.1 Verification as a Web-Service

In the last few years distributed applications over the WEB have gained wider popularity. The main advantages of exploiting the WEB as underlying platform can be summarized as follows. The WEB provides uniform mechanisms to handle computing problems which involve a large number of *heterogeneous* components that are *physically distributed* and *inter-operate* autonomously.

Recently, several software engineering technologies have been introduced to support a programming paradigm where the WEB is exploited as a *service distributor*. Rather than a monolithic application, a WEB server should be intended as a component available over the WEB that can possibly be exploited by others (human users or software applications) to develop new services. Conceptually, WEB services [102] are stand-alone components that reside over the nodes of the network and are network accessible through a network interface and standard protocols. Applications over the WEB are developed by combining and integrating WEB services together. WEB service has no pre-existing knowledge of what interactions with other WEB services may occur. Moreover, WEB services are highly portable and can easily be adapted to a variety of infrastructures.

In a WEB service scenario, the development of applications can be characterized in terms of the following steps:

1. *Publishing* WEB services;
2. *Finding* the required WEB services;
3. *Binding* the WEB services inside the application;
4. *Running* the application assembled from WEB services.

Indeed, in the next few years evolutionary in-development technologies based on HTTP/XML plus

1. remote invocation (e.g. XML-RPC SOAP),
2. directory and service binding (e.g. UDDI, trader),
3. language to express service features (e.g. WSDL)

will become the standard functional platform to programming applications over the WEB.

The vast majority of currently available semantic-based verification environments have been designed and implemented by sticking to traditional paradigms. Basically, verification environments are monolithic specialized servers which do not easily support interoperability and dynamic reconfiguration. We argue that the research activity in the field of formal verification can take advantage of the shift from the traditional development paradigms to other paradigms which better accommodate and support WEB services. We intend to explore the following issue:

> Can we simplify the design, development and maintenance of semantics-based verification environments in a modular fashion by exploiting WEB services?

A preliminary answer to this question is given by presenting the prototype version of a verification toolkit which directly exploits the WEB as a service distributor. The toolkit has been conceived to support reasoning about the behaviour of mobile systems specified as π-calculus processes and it supports the dynamic integration of several verification techniques.

Finally, the toolkit has been developed by targeting also the goal of extending an available verification environment (HAL [70, 71]) with new facilities provided as WEB services. This has given us the opportunity to verify the effective power of the WEB service approach to deal with the reuse and integration of "old" modules.

14.2 Preliminaries: HAL

HD-automata provide a finite state, finite branching representation of the behaviour of name passing calculi. The finiteness property given by the HD-automata has been exploited to automatize the check of behavioral properties. Indeed, a semantic-based verification environment for the π-calculus, called *HD Automata Laboratory* (HAL) [70, 71] has been implemented and experimented. HAL is written in C++ and compiled with the GNU C++ compiler (the GUI is written in Tcl/Tk), and runs on SUN stations (under SUN-OS).

HAL supports verification of logical formulae expressing properties of the behaviour of π-calculus agents. The construction of the HAL model checker facility has been done in two stages. First a high level logic with modalities indexed by π-calculus actions has been introduced and then a mapping which translates logical formulae into a classical modal logic for standard automata has been defined. The distinguished and innovative feature of the approach is that translation mapping is driven by the finite state representation of the system (the π-calculus process) to be verified.

HAL has been used to perform the verification of several case studies as, for example, the GSM handover protocol [143]. However, a main limitation of the current implementation of HAL is due to the state explosion problem that arises when dealing with real systems. A way to overcome this problem is to extend the environment with a minimization facility which provides the minimal HD-automata of a given π-calculus processes.

The work reported in [73] tackles the problem of minimizing LTS for name passing calculi in the abstract setting of coalgebraic theories. The main result of the paper is to provide a concrete representation of the terminal coalgebra giving the minimal HD-automaton.

14.3 Service Coordination

This section describes the issues related to the development of a verification toolkit which exploits the WEB as a service distributor. Here, we consider only two services, namely HAL and Mihda; however the same techniques can be exploited to integrate in a modular fashion a variety of services. The fundamental techniques which enables the dynamic integration of services is the separation between the service facilities (what the service provides) and the mechanisms that coordinate the way services interact. The main advantage of this approach consists of making service coordination usable in the context of formal verification.

HAL and Mihda provide several functionalities. The main issue to face with is the definition of WEB interfaces that make these toolkits accessible and usable on the Internet. This is done into two steps:

1. the first step defines the *WEB coordination interface* which, independently from the implementation technologies, describes the WEB interaction capabilities. In other words, the WEB coordination interface describes what a service can do and how to invoke it;

2. the second step transforms the program facilities which correspond to publish the coordination interface on the WEB.

The main programming construct we exploit to program service coordination is XML-RPC. XML-RPC is a protocol that defines a way to perform remote procedure calls using HTTP as underlying communication protocol and XML for encoding data. XML-RPC ensures interoperability among components available over the WEB at the main cost of parsing and serializing XML documents.

14.3.1 Service Creation

In our running example, the WEB coordination interface of Mihda provides three interaction capabilities: `compile`, `reduce` and `Tofc2`. The first interaction capability takes a π-calculus agent as input and yields as output the corresponding HD-automaton. The capability `reduce` performs minimization. Finally, the capability `Tofc2` transforms the Mihda representation of HD-automata into the FC2 format used inside HAL. The WEB coordination interface of HAL provides the `check` capability to perform model checking, the capability `unfold` which generates a standard automaton out of an HD-automaton, and the capability `visualize` allowing to graphically operate over HD-automata.

The publication on the WEB of the coordination interfaces has been performed by exploiting the facilities of Zope that is a web application server; it provides mechanisms to "publish" information on the WEB. However, Zope is much more. Indeed, Zope provides a comprehensive framework for management of web contents ranging from simple HTML pages to complete components. In particular, through Zope

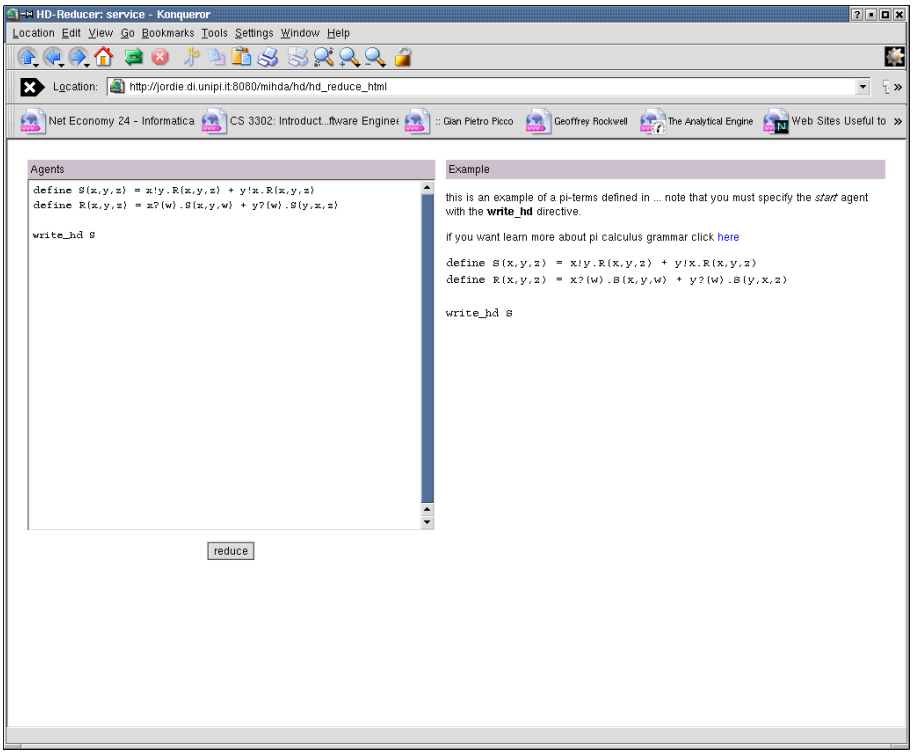


Figure 14.1: Mihda WEB Service

mechanisms the calls to the capabilities of the coordination interface are dynamically transformed into calls of the corresponding programs (e.g. via XML-RPC). Figure 14.1 illustrates the WEB interface of Mihda as provided by the Zope implementation.

14.3.2 Programming Service Coordination

In our experiment, the service coordination language is python, an interpreted object oriented scripting language which is widely used to connect existing components together. Expressiveness of python gives us the opportunity of programming service coordination in the same way traditional programming languages makes use of software libraries. In particular, services are invoked exactly as "local" libraries and all the issues related to data marshaling/unmarshalling and remote invocation are managed by the XML-RPC support.

An example of service coordination is illustrated in Figure 14.2 to verify a property of a specification, i.e. to test whether a π-calculus agent A is a model for a formula F. We can briefly comment on the coordination code of Figure 14.2. First, XML-RPC connections with the Mihda server and with HAL server are created and respectively recorded in variables `mihda` and `hal`. Then, a service of Mihda is invoked. More precisely, the result of executing the service `compile` is stored in the

```
from xmlrpclib import *
import sys

try:
    mihda = Server( "http://jordie.di.unipi.it:8080/mihda/hd" )

    hal = Server( "http://bladerunner.iei.pi.cnr.it:8080/hal" )

    hd = mihda.compile( A )

    reduced_hd = mihda.reduce( hd )

    reduced_hd_fc2 = mihda.Tofc2( reduced_hd )

    aut = hal.unfold( reduced_hd_fc2 )

    if hal.check( aut, F ):
        print 'ok'
    else:
        print 'ko'

except Exception, e:
    print "*** error ***"
```

Figure 14.2: Orchestrating HAL and Mihda services

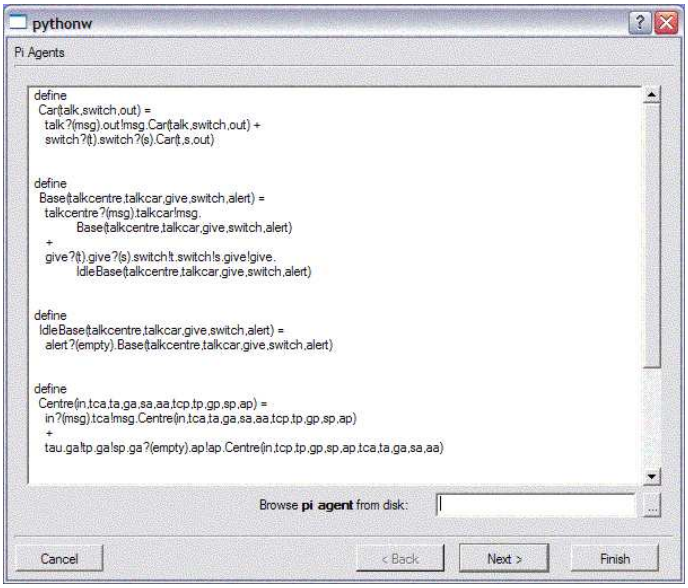


Figure 14.3: Compiling

variable `hd`.

Next, `hd` is minimized, by invoking the service `reduce` of Mihda; and, by applying the Mihda service `Tofc2`, the minimal automaton is transformed into the FC2 format. Variable `reduced_hd_fc2` contains a HD-automaton in a format suitable for being processes by the HAL service `unfold` that generate an ordinary automaton from a HD-automaton represented in FC2 format.

Finally, a message on the standard output is printed. The message depends on whether π-calculus agent A satisfies formula F or not. This is obtained by invoking the HAL model checking facility `check`. Notice that the coordination code can transparently handle both local and remote exceptions.

Figure 14.3 and Figure 14.4 illustrate the compiling of the π-calculus process specifying the GSM handover protocol, and the minimization step. Notice that the service coordination program runs under WindowsXP, thus pointing out the interoperability nature of the toolkit. Indeed, we recall that Mihda is written in ocaml and runs over linux machines, HAL is a GNU C++ application executed in SUN-OS and both are used by executing the code in Figure 14.3

We remark that the only part of the coordination code in Figure 14.2 that includes network dependencies is

```
mihda = Server( "http://jordie.di.unipi.it:8080/mihda/hd" )

hal = Server( "http://bladerunner.iei.pi.cnr.it:8080/hal" )
```

namely, the operation that opens connections with the HAL and Mihda servers. However, this network dependency can be removed by introducing a further module, namely the *directory of services* together with a simple *trader* facility. A directory

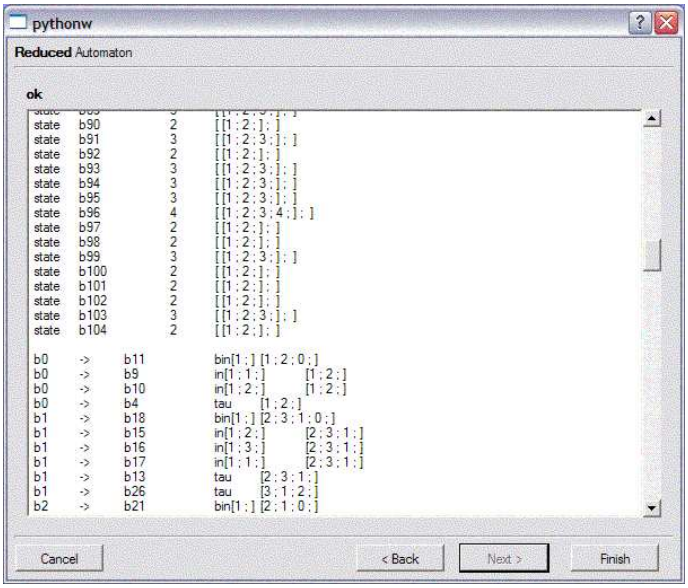


Figure 14.4: Minimizing

of services is a structure that maps the description of the WEB services represented by suitable types into the corresponding network addresses. Moreover, the directory of services performs the binding of services. In other words, the directory of services can be thought of as being a sort of enriched DNS for WEB services. The directory has two facilities. The `publish` facility is invoked to make available WEB service. The `query` facility which is used by applications to discover which are the available services. Hence, the `trader` can be used to obtain a WEB service of a certain type and to bind it inside the application.

The directory of services and the trader allow us to avoid specifying the effective names (and localities) of services into the source code and to dynamically bind services during the execution only on demand. Moreover, this mechanism makes transparent the distribution of services: when writing the coordination code the programmer is not aware of the localities of services. Hence, a service can also be replicated or re-allocated into a new locality without requiring any change into service coordination programs.

In our running example, to use a trader it is sufficient to substitute the assignments to `mihda` and `hal` variables with the following code:

```
import Trader

offers = Trader.query( "reducer/mihda" )

mihda = offers[ 0 ]  # choose the first

offers = Trader.query( "hal" )
```

```
hal = offers[ 0 ]  # choose the first
```

The invocation of the `query` procedure of the `Trader` library yields the list of services that match the parameter (i.e. the string describing the kind of services we are interested in).

Directories and traders permits to hide network details in the service coordination code. A further benefit is given by the possibility of replicating the services and maintaining a standard access modality to the WEB services under coordination. For instance, by substituting the assignment to `offers` in the previous code with

```
offers = Trader.query( "reducer" )
```

we obtain a polymorphic coordination code that, at run-time, is able to find, bind and finally invoke any service registered as "reducer".

14.4 Lessons Learned

We started our experiment with the goal of understanding whether the WEB service metaphor could be effectively exploited to develop in a modular fashion semantic-based verification environments. In this respect, the prototype implementation of a toolkit supporting verification of mobile processes specified in the π-calculus is a significative example.

Our approach adopts a service coordination model whose main advantage resides in reducing the impact of network dependencies and of dynamic addition/removal of WEB services by the well-identified notions of directory of services and trader. To the best of our knowledge, this is the first verification toolkit that specifically addresses the problem of exploiting WEB services.

The service coordination mechanisms presented in this paper, however, have some disadvantages. In particular, they do not exploit the full expressive power of SOAP to handle types and signatures. For instance, the so called "version consistency" problem (namely the client program can work with one version of the service and not with others) can be solved by types and signatures.

SOAP is well integrated inside the .NET framework which provides other powerful mechanisms to deal with types and metadata (i.e. description of types). In particular, metadata information can be extracted from programs at run time, and supplied to the *emitter* to generate the corresponding data structures together with their operations. Furthermore, the *Just-in-time* compiler turns them into native code. We plan to investigate and experiment the .NET framework to design "next generation" semantic-based verification environments.

Part IV

Concluding Remarks and Future Directions

Chapter 15

Conclusion and Future Work

Abstract

This chapter collects final remarks on this dissertation. In previous chapters we have discussed both foundational and experimental techniques that tackle some aspects related to distributed programming in the WAN setting.

This chapter is divided into three sections, one for each part of the thesis. Each section reports some final comments on the topics investigated in the corresponding parts of the dissertation. Hints on possible connections among these topics are also given.

Contents

15.1 A Declarative WAN Model

A first contribution of Part I is the definition of a declarative programming style based on hypergraphs. The intuition is that network components can be represented as hyperedges (edges connected to more than two nodes). Hyperedges exhibit their requirements by imposing constraints at their synchronization interfaces. A synchronization interface of an hyperedge is the set of nodes connected to the hyperedge. Nodes can be viewed as control points at which synchronization can take place. Computation is modeled via synchronized hyperedge replacement. Namely, adjacent edges are replaced as declared in their productions provided that their constraints are mutually satisfied.

We believe that our hypergraphical calculus captures some basic aspects of networks and distributed systems because it explicitly expresses synchronization points and also model mobility via a name passing mechanism.

We have applied hypergraphs to describe well-known models of WAN programming, namely Ambient and KLAIM. In particular we have studied how Ambient and KLAIM can be encompassed within the hypergraphical calculus.

The second contribution of Part I is the specification of QLAIM, an extension of KLAIM with primitives for specifying network connections and attributes over them. In QLAIM service agreements can be declared and dealt as first class citizen.

We have proposed two translations that respectively map Ambient and QLAIM into hypergraphs. Both mappings preserve the intuitive semantics of the calculi and give use the opportunity of comparing the underlying models. More precisely, process calculi that aim at modeling distributed computations at the foundational level usually represent networks as bunches of somehow connected sites where computations take place. Ambient and QLAIM are not an exception in this sense, because ambients and localities are used in Ambient and QLAIM, respectively, for representing network sites. Both ambients and localities are "passive" with respect to the computation and may contain information that are exploited by processes which are the only active entities. The hypergraphs which are images of Ambient or QLAIM nets are composed of two kind of edges. A first type of edges is strictly related to active entities in the source calculi. However, coordination productions are also specified for such edges. In the hypergraphs for Ambient those productions synchronize with productions of edges representing ambients. Hence, ambients are mapped in active component of the hypergraphical calculus, namely edges. Even more explicitly, edges $\mathfrak{S}$ and Δ in the encoding of QLAIM play the rôle of (active) coordinators. This observation sheds light on the fact that sites have a coordination rôle which is only implicit in process calculi models.

Another interesting observation is that the coordination mechanisms required for "implementing" Ambient and QLAIM are based on multi-parties synchronizations. For instance, *in* or *out* capabilities of Ambient require a synchronization between two ambients and the "pilot" process that fires the capability. A similar phenomenon arise when route discovery and reservation is set up in QLAIM, because a router edge

Δ must synchronize with its site edge $\mathfrak{S}$ and its gateway edges.

Finally, in the line of [97], we exploits the hypergraphical calculus for describing architectural design in dynamically reconfigurable environments. Indeed, we showed a methodology for representing and refining UML specifications with the further benefit that the hypergraph representation accounts for distribution related issues even though they were not considered into the initial UML specifications.

15.2 Security

Security is a mandatory concern when WAN applications are under investigation. First it must be said that security is a *multiform* concept with a deep impact on both WAN applications and their computations at any level. Moreover, at each level, security dresses different clothes, going from the access control mechanisms generally advocated at application level to inter-router authentication and secrecy certification at IPsec level. Part II focuses on cryptographic protocols and security properties as authentication, secrecy and integrity. We define the *c*ryptographic *I*nteraction *P*attern calculus (cIP) and $\mathcal{PL}$, a logic for expressing relationships among variable of cIP. The cIP calculus is basically a variant of π-calculus with cryptographic mechanisms. Principals of security protocols are modeled by means of cIP processes. One of the main benefits of cIP is that multi-sessions of protocols can be easily handled. In other words, protocols are expressed in a declarative manner by specifying each rôle of the protocol as a cIP process. Then cIP semantics automatically instantiates instances of the rôles and adds them in a running context. The symbolic semantics of cIP allows us to define a framework where protocols can be model checked with respect to security properties expressed as $\mathcal{PL}$ formulae. An implementation of the cIP framework is under development. We conjecture that multi-session attacks can be verified more easily in our framework, because both the specification of principals and security properties do not change with the number of instantiated rôles: They are declared once and for all at "declaration time".

Another interesting direction would be to exploit cIP framework for constraining coordination mechanisms of open systems. Indeed, hitherto, the join operation used in cIP to connect running contexts and new instance of principals is a non-deterministic primitive of the calculus. We can equip processes with formulae (e.g. $\mathcal{PL}$ formulae) that impose requirements on the possible contexts that the processes will join. Analogously, contexts can be equipped with formulae that constraints the possible participants (or the way they must be connected), as well. With this extension the join mechanism becomes a powerful coordination mechanism that accounts for security issues. Some initial results in this direction have appeared in [28]. Let us remark the strong connection between this coordination mechanism and synchronized hyperedge replacement mechanism defined for the hypergraphical calculus of Part I.

15.3 Verification Environment

Part III describes Mihda, a verification environment based on automata minimization. The environment is parameterized with respect to different classes of automata and equivalence relations between states.

One of the main results of this part is that, the correctness of the implementation is derived from the (declarative) co-algebraic specification. Indeed, the proof exploits the strict correspondence between theoretical concepts and the concrete structures of the implementation.

Another feature of Mihda resides in its modularity. Indeed, it has been designed with the aim of being an environment for modular minimization achieved through the partition refinement algorithm. The main architectural and implementation choices that feature this modularity have been pointed out. In this framework Mihda has been experimented for minimizing History Dependent Automata which result from π-calculus. However, Mihda has also been equipped with modules for minimizing ordinary automata according to the usual bisimulation relation.

Finally, we show how Mihda can be integrated with existing verification environments via a web interface that allows one to easily write simple programs that use a web interface to the verification facilities as a normal library.

A promising line of research lies in the definition of functors and their corresponding concrete structures for other semantics of π-calculus. In particular, we intend to investigate how the *open* bisimulation of the calculus can be coalgebraically defined and implemented in Mihda. This is of great interest for verification purposes because of the intrinsic *symbolic* flavour of the open semantics.

Related to this point is the current work we are conducing on the definition of a calculus based on the Fusion calculus of [146] for specifying a general theoretical framework that encompasses different variant of π-calculus and their semantics.

It would also be interesting to investigate possible relationships between the hypegraphical calculus and HD-automata. Indeed, if a suitable representation of hypergraphs can be given in terms of HD-automata, we could also exploits hypergraphs as an intermediate language that can be inserted into Mihda for minimizing systems specified within different frameworks. A possible way of achieve this result is by exploiting the encoding graphs into Petri nets introduced in [12, 15, 14] and the encoding of Petri Nets into HD-automata studied in [151].

Perhaps a more ambitious direction is the integration within a unique framework of partition refinement and "on the fly" techniques. This issue is probably related also to modular verification techniques. For instance, we can imagine of cases where we proceed using on-the-fly techniques and, when conditions for minimizing (part of) the automata using the partition refinement algorithm hold, we proceed with a minimization algorithm. This is a non trivial task and would require many theoretical efforts.

Bibliography

[1] Martín Abadi. Security protocols and specifications. In *FOSSACS: International Conference on Foundations of Software Science and Computation Structures*. LNCS, 1999.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[3] Peter Aczel. Algebras and coalgebras. In Roy Backhouse, Roland Crole and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, chapter 3, pages 79–88. Springer Verlag, April 2002. Revised Lectures of the Int. Summer School and Workshop.

[4] Peter Aczel and Nax Mendler. A final coalgebra theorem. In Lecture Notes in Computer Science, editor, *Category Theory and Computer Science*, volume 389, 1989.

[5] Jirí Adámek, Stefan Milius, and Jirí Velebil. Final coalgebras and a solution theorem for arbitrary endofunctors. In Lawrence S. Moss, editor, *Coalgebraic Methods in Computer Science*, Grenoble, France, April 2002.

[6] Robert Allen and David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proceedings of the IFIP 12th World Computer Congress. Volume 1: Algorithms, Software, Architecture*, pages 134–141, Amsterdam, The Netherlands, September 1992. Elsevier Science Publishers.

[7] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π–calculus. In U. Montanari and V. V. Sassone, editors, *Proc. of CONCUR'96*, volume 1119 of *LNCS*, pages 147–162. Springer, 1996.

[8] Luis Filipe Andrade and José Luiz Fiadeiro. Coordination for orchestration. In *COORDINATION 2002*. Lecture Notes in Computer Science, 2002.

[9] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1997.

[10] Ken Arnold, Ann Wollrath, Brian O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.

[11] Boumediene Bal, Henri E. Belkhouche and Luca Cardelli, editors. *Workshop on Internet Programming Languages*, volume 1686 of *LNCS*. Springer, 1999.

[12] Paolo Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars*. PhD thesis, Università di Pisa, Dipartimento di Informatica, March 2000.

[13] Paolo Baldan, Andrea Bracciali, and Roberto Bruni. Bisimulation by unification. In C. Ringeissen H. Kirchner, editor, *9th International Conference on Algebraic Methodology And Software Technology (AMAST'2002)*, volume 2422 of *Lecture Notes in Computer Science*, pages 254–270, Reunion Island, 2002. Springer Verlag.

[14] Paolo Baldan, Andrea Corradini, and Ugo Montanari. History preserving bisimulation for contextual nets. In Didier Bert, Christine Choppy, and Peter Mosses, editors, *WADT*, volume 1827, pages 291 – 310. Springer Verlag, 1999.

[15] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Bisimulation equivalences for graph grammars. In Wilfried Brauer, Juhani Karhumaeki, Arto Salomaa, and Hartmut Ehrig, editors, *Festschrift in Honor of Grzegorz Rozeberg*, volume 2300 of *Lecture Notes in Computer Science*, pages 158–187. Springer Verlag, 2002.

[16] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[17] Lorenzo Bettini, Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. Interactive mobile agents in XKlaim. In *[49]*, pages 110–115. IEEE Computer Society Press, 1998.

[18] Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. Structured nets in klaim. In *Proc. of the ACM SAC'2000, Special Track on Coordination Models, Languages and Applications*, pages 174–180. ACM Press, 2000.

[19] Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. Infrastructure language for open nets. In Proc. of the 2002 ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications. ACM Press, 2002. Special Track on Coordination Models, Languages and Applications.

[20] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[21] Grady Booch, Ivar Jacobson, James Rumbaugh, and Jim Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[22] Michele Boreale. Symbolic trace analysis of cryptographic protocols. In *28th Colloquium on Automata, Languages and Programming (ICALP)*, LNCS. Springer, July 2001.

[23] Michele Boreale and Marzia Buscemi. A framework for the analysis of security protocols. In *CONCUR: 13th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2002.

[24] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[25] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. Observing localities. *Theoretical Computer Science*, 114(1):31–61, June 1993.

[26] Andrea Bracciali, Antonio Brogi, Gianluigi Ferrari, and Emilio Tuosto. A magic approach to the analysis of security protocols. Available at `http://www.di.unipi.it/~etuosto/`.

[27] Andrea Bracciali, Antonio Brogi, Gianluigi Ferrari, and Emilio Tuosto. Security issues in component based design. In *ConCoord Workshop 2001, Lipari - Italy*, 2001.

[28] Andrea Bracciali, Antonio Brogi, Gianluigi Ferrari, and Emilio Tuosto. Security and dynamic compositions of open systems. In *PDPTA 2002*, pages 1372 – 1377, 2002.

[29] Andrea Bracciali, Antonio Brogi, and Franco Turini. Coordinating interaction patterns. In *Proceedings of the* ACM Symposium on Applied Computing*, Las Vegas, USA*. ACM, 2001.

[30] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Reasoning about security in mobile ambients. In *Concur 2001*, number 2154 in LNCS, pages 102–120. Springer, 2001.

[31] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[32] Luis Caires and Luca Cardelli. A spatial logic for concurrency (Part I). In *TACAS*, Lecture Notes in Computer Science. Springer Verlag, 2001.

[33] Luis Caires and Luca Cardelli. A spatial logic for concurrency (Part II). *Information and Computation*, 2001.

[34] Luis Caires and Luca Cardelli. A spatial logic for concurrency (Part II). In *CONCUR'02*, volume 2421 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[35] Luca Cardelli and Rowan Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.

[36] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. *Lecture Notes in Computer Science*, 2380, 2002.

[37] Luca Cardelli and Giorgio Ghelli. A query language based on the ambient logic. In David Sands, editor, *Programming Languages and Systems: Proceedings of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 1–22, Genova, Italy, April 2001. Springer. Held as Part of the Joint European Conferences on Theory and Practice of Software.

[38] Luca Cardelli and Andrew D. Gordon. A commitment relation for the ambient calculus. Manuscript.

[39] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *TCS: Theoretical Computer Science*, 240, 2000.

[40] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[41] Nicholas Carriero, David Gelernter, and Lenore Zuck. Bauhaus Linda. *Lecture Notes in Computer Science*, 924:66–76, 1995.

[42] Giuseppe Castagna, Giorgio Ghelli, and Francesco Zappa-Nardelli. Typing mobility in the seal calculus. In *International Conference on Concurrency Theory*, pages 82–101, 2001.

[43] Ilaria Castellani and Ugo Montanari. Graph Grammars for Distributed Systems. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag, 1983.

[44] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In Paul Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 35–51, Cambrige, UK, July 2000. IEEE Computer Society Press.

[45] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000. ISBN 2-84177-121-0.

[46] Tsu-Wei Chen, Mario Gerla, and Jack Tzu-Chieh Tsai. QoS routing performance in multihop wireless networks. In *ICUPC97*, San Diego, 1997. IEEE.

[47] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.html, March 2001. W3C Note.

[48] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[49] Antonio Corradi, Andrea Omicini, and Agostino Poggi, editors. *Proc. of 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE) '98*. IEEE Computer Society Press, 1998.

[50] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Raiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In Rozenberg [158], chapter 3.

[51] Scott Corson and Josheph P. Macker. *Mobile Ad Hoc Networking*, January 1999. RFC 2501.

[52] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In *Mobile Object Systems: Towards the Programmable Internet*, pages 93–110. Springer-Verlag, 1997.

[53] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North-Holland, 1958.

[54] Mads Dam. Model Checking Mobile Processes. *Information and Computation*, 129(1):35–51, 1996.

[55] Nikolas G. de Bruijn. A survey of the project automath. In J. P. Seldin and J. R. Hindley, editors, *H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[56] Rocco De Nicola, Gianluigi Ferrari, Ugo Montanari, Rosario Pugliese, and Emilio Tuosto. A formal basis for reasoning on programmable qos. In *International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[57] Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[58] Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. Types as specifications of access policies. *[170]*, 1603:117–146, 1999.

[59] Rocco De Nicola, Gianluigi Ferrari, Rosario Pugliese, and Betti Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, June 2000.

[60] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[61] Pierpaolo Degano and Ugo Montanari. A model of distributed systems based of graph rewriting. *Journal of the ACM*, 34:411–449, 1987.

[62] Jo lle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax with induction in coq. In Frank Pfenning, editor, *5th International Conference on Logic Programming and Automated Reasoning*, number 822 in LNAI, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag.

[63] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[64] Whitfield Diffie and Martin Hellman. New directions in cryptography. In *SIMMONS: Secure Communications and Asymmetric Cryptosystems*, 1982.

[65] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEETIT: IEEE Transactions on Information Theory*, 29, 1983.

[66] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[67] Gregor Engels, Jan Hendrik Hausmann, Raiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioural diagrams in UML. In *UML 2000*, number 1939 in Lecture Notes in Computer Science, pages 323–337. Springer, 2000.

[68] Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. Honest ideals on strand spaces. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW '98)*, pages 66–78, Washington - Brussels - Tokyo, June 1998. IEEE.

[69] Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.

[70] Gianluigi Ferrari, Giovanni Ferro, Stefania Gnesi, Ugo Montanari, Marco Pistore, and Gioia Ristori. An automata based verification environment for mobile processes. In Ed Brinksma, editor, *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97),*

Enschede, The Netherlands, volume 1217 of *LNCS*, pages 275–289. Springer, April 1997.

[71] Gianluigi Ferrari, Stefania Gnesi, Ugo Montanari, Marco Pistore, and Gioia Ristori. Verifying mobile processes in the HAL environment. In *Proc. 10th International Computer Aided Verification Conference*, pages 511–515, 1998.

[72] Gianluigi Ferrari, Stefania Gnesi, Ugo Montanari, Roberto Raggi, Gianluca Trentanni, and Emilio Tuosto. Verification on the web. Submitted for publication.

[73] Gianluigi Ferrari, Ugo Montanari, and Marco Pistore. Minimizing transition systems for name passing calculi: A co-algebraic formulation. In Mogens Nielsen and Uffe Engberg, editors, *FOSSACS 2002*, volume LNCS 2303, pages 129–143. Springer Verlag, 2002.

[74] Gianluigi Ferrari, Ugo Montanari, and Paola Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1):53–103, November 1996.

[75] Gianluigi Ferrari, Ugo Montanari, Roberto Raggi, and Emilio Tuosto. From coalgebraic specification to toolkit developement. Submitted for publication.

[76] Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto. A lts semantics of ambients via graph synchronization with mobility. In *7th Italian Conference on Theoretical Computer Science – ICTCS'01*, volume 2202 of *LNCS*. Springer, 2001.

[77] Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto. Graph-based models of internetworking systems. In A. Haeberer, editor, *Formal Methods at the Crossroads: from Panaces to Foundational Support*, Lecture Notes in Computer Science. Springer, 2003. To appear.

[78] Gianluigi Ferrari, Carlo Montangero, Laura Semini, and Simone Semprini. Mark: A reasoning kit for mobility. *Automated Software Engineering*, 9(2):137–150, 2002.

[79] Riccardo Focardi and Roberto Gorrieri. A classification of security properties. *Journal of Computer Security*, 3(1), 1995.

[80] Riccardo Focardi and Roberto Gorrieri. The Compositional Security Checker: A tool for the verification of information flow security properties. *IEEE*, 23(9):550–571, September 1997.

[81] Cedric Fournet and George Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996.

[82] Cedric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile processes. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag.

[83] Cedric Fournet, Jean-Jacques Lévy, and Alain Schmitt. A distributed implementation of ambients. Available at `http://join.inria.fr/ambients.html`.

[84] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol — version 3.0. Available at `http://home.netscape.com/eng/ssl3/ssl-toc.html`, March 1996.

[85] Fabio Gadducci and Ugo Montanari. A concurrent graph semantics for mobile ambients. In Stephen Brooks and Michael Mislove, editors, *Electronic Notes in Theoretical Computer Science*, volume 45. Elsevier Science Publishers, 2001.

[86] David Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[87] David Gelernter. Multiple tuple spaces in linda. *Parle 89*, page 1, March 1989.

[88] Martin Gogolla. Graph transformations on the UML Metamodel. In *ICALP Workshop on Graph Transformations and Visual Modeling Techniques*, pages 359–371. Carleton Scientific, 2000.

[89] Dieter Gollman. *Computer Security*. John Wiley & Sons, 1999.

[90] Ursula Goltz and Wolfgang Reisig. The non-sequential behaviour of petri nets. *Information and Computation*, 57(2/3):125–147, 1983.

[91] Kevin John Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, 2002. ISBN 0-13-062296-6.

[92] Dan Harkins and Dave Carrel. RFC 2409: The Internet Key Exchange (IKE), November 1998. Status: PROPOSED STANDARD.

[93] Mattew Hennessy and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *entcs*, pages 3–17. Elsevier Science Publishers, 1998. Full version as CogSci Report 2/98, University of Sussex, Brighton.

[94] Mattew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. *[170]*, pages 95–115, 1999.

[95] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *27th International Colloquium on Automata, Languages and Programming (ICALP '2000)*, July 2000. A longer version appeared as Computer Science Technical Report 2000:03, School of Cognitive and Computing Sciences, University of Sussex.

[96] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Reconfiguration of software architecture styles with name mobility. In Antonio Porto and Gruia-Catalin Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer Verlag, 2000.

[97] Dan Hirsch and Ugo Montanari. Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In *12th International Conference in Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 121–136, Aalborg, Denmark, 2001. Springer Verlag.

[98] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. & 0-13-153289-8.

[99] Koei Honda and Mario Tokoro. An object calculus for asynchronous communication. *Lecture Notes in Computer Science*, 512:133–147, 1991.

[100] William A. Howard. The formulae-as-type notion of construction. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

[101] Antti Huima. Efficient finite-state analysis of security protocols. In *Formal methods and security protocols*, FLOC Workshop, Trento, 1999. INRIA.

[102] IBM Software Group. Web services conceptual architecture. In *IBM White Papers*, 2000.

[103] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62:222–259, 1996.

[104] Bengt Jonsson and Joachim Parrow. Deciding bisimulation equivalences for a class of Non-Finite-State programs. *Information and Computation*, 107(2):272–302, December 1993.

[105] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

[106] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, pages 5 – 38, January and 161 – 191, February, 1883.

[107] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A formal model for role-based access control using graph transformation. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *ESORICS*, volume 1895 of *Lecture Notes in Computer Science*, pages 122–139, 6th European Symposium on Research in Computer Security, 2000. Springer Verlag.

[108] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Foundations for a graph-based approach to the specification of access control policies. In Furio Honsell and Marina Lenisa, editors, *FoSSaCS*, Lecture Notes in Computer Science, Foundations of Software Science and Computation Structures, 2001. Springer Verlag.

[109] Manuel Koch and Francesco Parise-Presicce. Describing policies with graph constraints and rules. In Andrea Corradini, Hartmut Ehrig, Hans-J org Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 223–238, First International Conference on Graph Transformation, Barcelona, Spain, October 2002. Springer Verlag.

[110] Barbara Koenig and Ugo Montanari. Observational equivalence for synchronized graph rewriting. In *Proc. TACS'01*, Lecture Notes in Computer Science. Springer Verlag, 2001. To appear.

[111] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. An Integrated Semantics for UML Class, Object, and State Diagrams based on Graph Transformation. In Michael Butler and Kaisa Sere, editors, *3rd Int. Conf. Integrated Formal Methods (IFM'02)*, Lecture Notes in Computer Science. Springer, Berlin, 2002.

[112] Peter Lee, Fritz Henglein, and Neil D. Jones, editors. *Proc. of the ACM Symposium on Principles of Programming Languages*. ACM Press, 1997.

[113] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 352–364, Boston, Massachusetts, January 2000.

[114] Gong Li. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, Reading, MA, USA, 1999.

[115] Gavin Lowe. A hierarchy of authentication specifications. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[116] Gavin Lowe. Towards a completeness result for model checking of security protocols. In *Proceedings of Computer Security Foundation Workshop*. IEEE, 1998.

[117] Gavin Lowe. Defining information flow. Technical report, Department of Mathematics and Computer Science, University of Leicester, 1999.

[118] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[119] Per Martin-L of. A theory of types. Technical Report 71-3, University of Stockholm, 1971.

[120] Per Martin-L of. *Intuizionistic Type Theory*. Bibliopolis, 1984.

[121] Peter J. McCann and Gruia-Catalin Roman. Compositional programming abstraction for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.

[122] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[123] Alfred J. Menzies, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[124] Microsoft Corporation. .NET: Ecma standards. Available at `http://msdn.microsoft.com/net/ecma`.

[125] Jonathan K. Millen. A necessarily parallel attack. In Nevin Heintze and Edmund M. Clarke, editors, *Workshop on Formal Methods and Security Protocols, Part of the Federated Logic Conference*, Trento, Italy, 1999.

[126] Robin Milner. *Communication and Concurrency*. Printice Hall, 1989.

[127] Robin Milner. *Communication and Concurrency*. Printice Hall, 1989.

[128] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[129] Robin Milner. *Commuticating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[130] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[131] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[132] John C. Mitchell, Mark Mitchell, and Ulrich Ster. Automated analysis of cryptographic protocols using murϕ. In *10th IEEE Computer Security Foundations Workshop, IEEE Press*, pages 141–151, 1997.

[133] John C. Mitchell, Mark Mitchell, and Ulrich Ster. Automated analysis of cryptographic protocols using murϕ. In *10th IEEE Computer Security Foundations Workshop*, pages 141–151. IEEE Press, 1997.

[134] Ugo Montanari and Marco Pistore. History dependent automata. Technical report, Computer Science Department, Università di Pisa, 1998. TR-11-98.

[135] Ugo Montanari and Marco Pistore. π-calculus, structured coalgebras, and minimal HD-automata. In Mogens Nielsen and Branislav Roman, editors, *MFCS: Symposium on Mathematical Foundations of Computer Science*, volume 1983 of *LNCS*. Springer Verlag, 2000. An extended version will be published on Theoretical Computer Science.

[136] Ugo Montanari and Francesca Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[137] National and Bureau of and Standards. *Data Encryption Standard*. U. S. Department of Commerce, Washington, DC, USA, January 1977.

[138] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[139] Tobias Nipkow and Lawrence C. Paulson. Isabelle. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *LNAI*, pages 673–676, Saratoga Springs, NY, June 1992. Springer Verlag.

[140] Object Management Group. Corba: Architecture and specification. Available at `http://www.omg.org`, 1998.

[141] Object Management Group. Unified modelling language specification, 2001.

[142] Oracle. Oracle 9*i*AS application server lite web page. In *http://www.oracle.com/*, 1999.

[143] Fredrik Orava and Joachim Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4(5):497–543, 1992.

[144] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In ACM, editor, *[112]*, pages 256–265, New York, NY, USA, 1997. ACM Press.

[145] Anthony S. Park and Peter Reichl. Personal Disconnected Operations with Mobile Agents. In *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, 1998.

[146] Joachim Parrow and Bjorn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *13th Annual IEEE Symposium on Logic and Computer Science*. IEEE Computer Society Press, 1998.

[147] Lawrence C. Paulson. *The inductive approach to verifying cryptographic protocols*. Technical report; no. 443. 4006797499. University of Cambridge Computer Laboratory, Cambridge, UK, USA, February 1998.

[148] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[149] Cyrus Peikari and Seth Fogie. *Windows .NET Server Security Handbook*. Prentice Hall, 2002. ISBN 0-13-047726-5.

[150] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press. Also available as Technical Report WUCS-98-21, July 1998, Washington University in St. Louis, MO, USA.

[151] Marco Pistore. *History dependent automata*. PhD thesis, Computer Science Department, Università di Pisa, 1999.

[152] Gordon G. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981.

[153] Paola Quaglia. *The π-calculus with explicit substitutions*. PhD thesis, Università di Pisa, Dipartimento di Informatica, 1996.

[154] James Riely and Mattew Hennessy. Trust and Partial Typing in Open Systems of Mobile Agents. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages (POPL'99)*, pages 93–104, 1999.

[155] James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 266(1–2):693–735, 2001.

[156] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978.

[157] Gruia-Catalin Roman, Peter J. McCann, and J. Y. Plunn. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.

[158] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.

[159] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[160] Ryan and Schneider. Process algebra and non-interference. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[161] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[162] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.

[163] Steve Schneider. Modelling security properties with CSP. Technical Report CSD-TR-96-04, Department of Computer Science, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, February 1996.

[164] Steve Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[165] Peter Sewell. From rewrite rules to bisimulation congruences. *Lecture Notes in Computer Science*, 1466, 1998.

[166] Mary Shaw and David Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[167] Douglas R. Stinson. *Cryptography: Theory and practice*. CRC Press, 1995.

[168] David Teller, Pascal Zimmer, and Daniel Hirschkoff. Using ambients to control resources. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *Concurrency Theory, 13th International Conference*, volume

2421 of *Lecture Notes in Computer Science*, Brno, Czech Republic, August 2002. Springer Verlag.

[169] Jan Vitek and Giuseppe Castagna. Towards a calculus of secure mobile computations. In *[11]*, Chicago, Illinois, May 1998.

[170] Jan Vitek and Christian D. Jensen. *Secure Internet programming: security issues for mobile and distributed objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1999.

[171] Dennis Volpano. Formalization and proof of secrecy properties. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[172] David Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, February 1995.

[173] Glynn Winskel. Synchronization trees. *Theoretical Computer Science*, May 1985.

[174] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages 2–12, Palm Springs, CA, USA, October 1999.

[175] James Worell. Terminal sequences for accessible endofunctors. In Lecture Notes in Computer Science, editor, *Category Theory and Computer Science*, volume 19, 1999.

[176] Tatu Ylonen. SSH — secure login connections over the Internet. In USENIX Association, editor, *6th USENIX Security Symposium, July 22–25, 1996. San Jose, CA*, pages 37–42, Berkeley, CA, USA, July 1996. USENIX.

[177] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

Cui legisse satis non est epigrammata centum
nil illi satis est, Caediciane, mali.
(Marziale, Lib. I, 118)

And then one day you find
Ten years have got behind you
No one told you when to run
You missed the starting gun

And you run, and you run to catch up with the sun, but it's sinking
Racing around to come up behind you again
The sun is the same in a relative way, but you're older
Shorter of breath and one day closer to death

Time (Roger Waters)
The Dark Side of the Moon - March 24th 1973