Behavioural patterns and software composition

Andrea Bracciali

May 2003

Addr: via F. Buonarroti 2, 56127 Pisa, Italy.
Tel:+39-50-2212743. Fax: +39-50-2212726.
E-mail: braccia@di.unipi.it Home-Page: http://www.di.unipi.it/~braccia
Thesis Supervisors:

Prof. Franco Turini and Prof. Antonio Brogi
Abstract

The way in which we conceive of software is substantially changing and our current experience of the digital world is based on distributed environments, populated by applications which consist of components interacting with each other, and tend to be distributed over different domains, if not moving throughout them. Other than computational skills, the success of an application depends even more on the capability of its components to successfully coordinate their tasks.

In this thesis we concentrate on the definition of a formal model to describe and verify the composition of the interactive behaviours of components. This study is motivated by the necessity to coordinate components, which are autonomous in their behaviours, and need to cooperate in an environment which may be only partially specified and subject to timely changes, since it is dynamically evolving, and possibly even populated by hostile components trying to exploit the work of the others.

Starting from a simplified name-passing calculus, we extend traditional component interfaces with behavioural descriptions, define the mechanisms by means of which components can be connected together, and characterise the open environment where they interact. A notion of correctness is defined for open environments and an algorithmic procedure for its verification is provided. Components spoiling the correctness of the system are not admitted. To release the last constraint, a methodology to adapt and hence accept components with mismatching behaviours is provided. The framework fully supports the verification of security protocols, when they are seen as a composition of principals in an open environment subject to malicious components. A complete methodology for formalising and verifying security protocols is presented.
Acknowledgments

During this work I confirmed to myself that research is a collective process, both in terms of building on the effort of others, and of being continuously influenced and inspired by the interaction with others. In this sense I would like to thank those who contributed to this work and to the improvement of my knowledge and my way of thinking, and more generally my personal views.

I would like to acknowledge the support I had at the Dipartimento di Informatica in developing my thesis. My supervisors, Franco Turini and Antonio Brogi, deserve my sincere gratitude. Franco Turini, despite his many duties, has been in these years a brilliant and friendly advisor, who taught me a lot. Antonio Brogi guided me in the development of many parts of this thesis, often correcting my errors and suggesting possible approaches to the problems addressed. We had long discussions which were demanding but satisfying at the end.

Gianluigi Ferrari, initially in charge of reviewing my work, has given to me several ideas and references, from which we found common research interests that produced a part of the work presented here. His friendly collaboration is greatly appreciated. An essential part in this collaboration has been played by Emilio Tuosto, with whom I shared the hard work and the rewards of searching for ways to solve problems (and going around trying to convince others that they can be interesting). I hope that what will be remembered most by Emilio is the satisfaction we shared, rather than the hard work. I also worked with Carlos Canal, from the University of Malaga, in an interesting and productive collaboration, and with Tommaso Bolognesi, with whom I collaborated at IEI-CNR.

I would like to thank the referees of the thesis, Farhad Arbab and Ernesto Pimentel, for having devoted part of their time to help me improve the work, and Paolo Mancarella, who gave me the opportunity of further developing it.

I am happy to have worked with Paolo Baldan and Roberto Bruni on related problems but not reported in this thesis, and I want to thank them for their friendship and for their instructive and rewarding collaboration.

I would also like to thank the many people who shared with me their experiences, not only professional ones, in these last years. Amongst them Marco, whom I met at the very beginning of my university life, Nadia, Alessandra and Simone, who made me welcome at the start of my PhD studies, and Diego, who is always able to see yet another alternative.
I hope my parents, Braccio e Vanna, may enjoy and be gratified by this work without the need of reading it, knowing that it is also thanks to them that the work has been done.

I want to apologize to Emily because of the stress that I may have passed on to her while I was carrying out this thesis, and thank her for the many suggestions she gave me about how to write properly and well (not all of which I am confident to have fully understood). But, mainly, I want to thank her for being as she is in her personal attitude to life.
## Contents

1 Introduction ............................... 1
   1.1 Motivations .................................. 1
   1.2 Contributions of the thesis ................. 5
      1.2.1 A model for component interaction ........ 5
      1.2.2 A model for component adaptation .......... 7
      1.2.3 A model for system security ............... 8
   1.3 Outline of the Thesis ....................... 10
   1.4 Origin of chapters ......................... 10
   1.5 Technical background ....................... 11
      1.5.1 Transition Systems ........................ 11
      1.5.2 The π-calculus ........................... 13
      1.5.3 Model checking for verification .......... 18

2 State of the art ........................... 19
   2.1 Coordination models and languages .......... 20
      2.1.1 Motivations ............................... 20
      2.1.2 Coordination models ....................... 21
      2.1.3 Discussion ................................ 24
   2.2 Software Architecture ........................ 25
      2.2.1 Motivations ............................... 25
      2.2.2 Architecture Description Languages ......... 26
      2.2.3 Future perspectives ....................... 29
      2.2.4 Discussion ................................ 30
   2.3 Component Based Software Engineering ...... 31
      2.3.1 Motivations ............................... 31
      2.3.2 Current trends ............................. 32
      2.3.3 Future perspectives ....................... 33

3 Interaction Patterns and Sessions ........... 35
   3.1 Components and their environment .......... 36
   3.2 Interaction Pattern calculus (IP-calculus) .... 40
      3.2.1 Interaction Pattern syntax ................ 40
      3.2.2 Interaction Pattern semantics .......... 44
3.3 Sessions ........................................ 47
  3.3.1 Accessing a session ......................... 49
  3.3.2 Session semantics ............................ 54

3.4 Correctness of sessions .......................... 58
  3.4.1 Correctness of closed sessions ............... 59
  3.4.2 Correctness of open sessions ................. 61

3.5 Verifying acceptability ......................... 65
  3.5.1 Completion standard form ..................... 66
  3.5.2 The algorithm $A$ (Acceptability checker) .... 72
  3.5.3 Constructing completions: examples ........... 79
  3.5.4 Complexity .................................. 82
  3.5.5 Correctness of the algorithm ................ 84

4 Adaptors ........................................... 93
  4.1 Motivations and overview ....................... 95
  4.2 Component interfaces ............................ 97
  4.3 Adaptor specification ........................... 99
  4.4 Adaptor derivation ............................. 104
  4.5 An example of adaptation ....................... 110
  4.6 About the role of the adaptor .................. 113
  4.7 Concluding remarks ............................. 116

5 Secure interaction in open systems ............... 119
  5.1 Cryptography + Security protocols = Security? .. 123
    5.1.1 Cryptography ................................ 124
    5.1.2 Security protocols and their verification ...... 125
    5.1.3 Security properties ........................... 128
    5.1.4 Discussion: $(c)IP$-calculus and security ....... 129
  5.2 The Intruder model .............................. 130
  5.3 cIP: A Cryptographic calculus of Interacting Principals ... 137
    5.3.1 Syntax ..................................... 138
    5.3.2 Sessions and Multiple-sessions ............... 141
    5.3.3 Operational semantics ......................... 143
  5.4 $\mathcal{P}L$ Logic: expressing security properties .... 150
    5.4.1 Syntax ..................................... 151
    5.4.2 Models for $\mathcal{P}L$ ......................... 153
  5.5 Concrete protocol verification ................... 155
  5.6 An Oracle discovers an attack .................. 158
  5.7 Symbolic protocol verification ................... 162
    5.7.1 A symbolic extension of cIP-calculus .......... 164
    5.7.2 Correctness of symbolic protocol verification .... 167
    5.7.3 Generating symbolic messages .................. 168
    5.7.4 Symbolic trace generation: An example .......... 174
5.7.5 Symbolic verification ............................... 176
5.7.6 Symbolic verification: An example ............... 179
5.8 Related work ........................................... 180
5.9 Concluding remarks ................................... 182
5.10 Proofs of Chapter 5 .................................. 184
  5.10.1 Proof of Theorem 5.3 [κ ⊃ m is decidable] ........ 185
  5.10.2 Proof of Theorem 5.4
    [∀ ψ Σ₀ ▼ ∼ₙ Σₙ ⇒ Σ₀ ▼→ₙ Σₙψ] ...................... 189
6 Conclusions .............................................. 191

Bibliography .............................................. 197

Index ....................................................... 211
List of Figures

1.1 Late transition system for the π-calculus. ................. 15
1.2 Early transition system for the π-calculus. ................. 16

3.1 Data and communication actions in IP-calculus. ............ 40
3.2 Behavioural expression syntax. .......................... 42
3.3 Structural equivalence axioms. .......................... 45
3.4 Interaction pattern semantics. .......................... 46
3.5 Session semantics. ...................................... 54
3.6 Acceptability is preserved by transitions. .................. 64
3.7 Algorithm A at work - I. ................................ 73
3.8 Completion construction. ................................ 77
3.9 Algorithm A at work - II. ................................ 80
3.10 Algorithm A at work - III. ................................ 81
3.11 Algorithm A at work - IV. ................................ 82

4.1 Adaptor construction. .................................... 108

5.1 Behavioural expressions of principals in cIP ................ 139
5.2 Principal semantics. ...................................... 144
5.3 Asynchronous session semantics. .......................... 146
5.4 Symbolic session semantics. .............................. 166
Chapter 1
Introduction

Open system is a term generically used, by different communities in the software development research area, to denote a software system consisting of dynamically interacting components. By discussing in more details the meaning of the term, we identify a set of unifying features that may be used to characterise some of the problems currently addressed by different approaches to software composition.

We then illustrate how some of these problems have been addressed in this thesis. A survey of the literature in the field will be presented in Chapter 2.

1.1 Motivations

Software design is evolving towards new architectures and the key aspect in this shift is the importance that interaction has assumed in the life of applications.

Applications no longer compute in isolation, but consist of components interacting with each other and with their environment. Besides computational skills, the success of an application depends on the capability of its components to successfully coordinate their tasks and interact with the environment.

The traditional interpretation of a computer program as an input-output transformation function no longer appears well suited for coping with the non-functional aspects of computation, like properties that may depend on the interactive behaviour of components.

Rarely, applications operate in an own domain, but rather, spread and distributed over networks, they may belong to different domains or depend on components beyond their control. The environment itself, hence, may be only partially accessible, in its being populated by other independent components which may dynamically provide and require services, or even try to maliciously exploit the work of the others.

This evolution of software development has been supported and fostered by the availability of large integration capabilities, both at the hardware and the software level. The wide connectivity made available by the development of network infras-
Other than data mobility, applications can also exploit mobility capabilities of the code, both to download programs as resources to be locally executed (code on demand) and to autonomously migrate, with or without their run-time environment, from one site to another.

These emerging scenarios, which envisage the pervasive diffusion of computational devices into every-day life, call for the definition of new computational models, motivating the development of new research areas.

Within the 2001 Research Programme of the European Commission, Information Society Technologies Programme (IST) [Com01], the initiative called Global Computing (Co-operation of Autonomous and Mobile Entities in Dynamic Environments) promotes the investigation of systems characterised as (emphasis added):

- “Systems composed of autonomous computational entities where activity is not centrally controlled, either because global control is impossible or impractical, or because the entities are created or controlled by different owners.

- The computational entities are mobile, due to the movement of the physical platforms or by movement of the entity from one platform to another.

- The configuration varies over time. For instance, the system is open to the introduction of new computational entities and likewise their deletion. The behaviour of the entities may vary over time.

- The systems operate with incomplete information about the environment. For instance, information becomes rapidly out of date and mobility requires information about the environment to be discovered.”

The initiative also recommends “to provide a solid scientific foundation for the design of such systems, and to lay the groundwork for achieving effective principles for building and analysing such systems.”

Systems with the above mentioned features, are generally called in the research community open systems. Let us briefly discuss their distinguishing features, as they emerge from the above presentation.

- AUTONOMY. Components may have their own control. Following the multi-agent terminology, components can exhibit pro-active behaviours in the sense that they can make autonomous choices in order to accomplish their tasks. The overall behaviour of a system, depends hence on the “coordination” of the behaviours of the components in the system. Even if some proposals read coordination as coercion, for example by wrapping components by a “controlling” code, in the general case, in which an application depends on the interaction with other components that do not belong to the same administrative domain, this cannot be achieved. Coordination hence must be read
as *orchestration* of the activities of the single components, which must be provided with suitable *coordination interfaces*, [AFG+02]. Coordination interfaces are only the first step towards the coordination of autonomous activities and its verification. The development of suitable coordination techniques, languages and models, clearly separating coordination from computation concerns [CG92, PA98, SBMW99], is currently an active research area, consisting of several communities that are trying to apply coordination to different contexts.

• **DYNAMICS (AND MOBILITY).** The dynamics of an open system may depend on several factors, like components that connect to or disconnect from the application, are dynamically reconfigured or migrate from one site to another one. The critical issue in these cases is represented by the needs of dynamic binding mechanisms, which make the traditional compilation-linking phase of application development out of date, requiring “just-in-time binding” mechanisms, [AFG+02]. It is important to observe that binding does not regard only the run-time environment, as usually understood, but also the connections among components and resources, so that the binding phase plays a decisive role in setting the possibilities of the interaction that will occur among components. It is straightforward, then, that any model aimed at the analysis of open systems cannot fail to consider an appropriate representation of binding mechanisms.

• **HETEROGENEITY.** Applications must be open with respect to components, data and resources that do not share a homogeneous programming paradigm. In particular, Component Based Software Engineering (CBSE) [BW98] is centered around the concept of “component as unity of deployment”, in order to allow the re-use of existing components. Components can be assembled together in a system, both statically, at the development time, and dynamically, in the sense that in a running system a component is added, upgraded, or possibly deleted without the needs of recompiling the whole system. Components, not necessarily developed within the same process of system assembly (COTS- commercial off the shelf - products, [SS]) are hence seen as black-boxes (often they are in the form of binary code) equipped with some description of their functionalities (often called *metadata*) which allows (or at least facilitates) their dynamic binding. An active direction of research regards the behavioural aspects of component composition.

Other attempts to overcome heterogeneity problems consist in the adoption of universally recognised description standards, or, more flexibly, of common description languages mediating among the different paradigms, see for instance the current interest about XML [BPSM97], as description language, and, for example, its use in the Web Service architecture [GGKS02], as service description language.
Heterogeneity also regards the problem of legacy software, as the number of existing applications that one wants to integrate in the new architectures, but that often have not been designed for this purpose.

Finally, it is worth pointing out that the problem of semantical heterogeneity, i.e. the problem of, possibly automatically, understanding the semantics of the services so as, for instance, to make a component able to automatically search, select and negotiate the services it needs, is an open research topic beyond the scope of this thesis.

- **INCOMPLETE ACCESSIBILITY OF THE ENVIRONMENT.** The partial accessibility of the environment, both because of domain boundaries or of its dynamical evolution, other than requiring adaptability capabilities to its components makes the analysis and the formal verification of open systems more complicated. Abstractions capable of dealing with evolving or partial specifications are hence needed, and correspondingly, a new interpretation of system correctness and other properties.

- **SECURITY.** A distributed domain exposes applications to the action of malicious components that can be out of control. It is a commonly accepted hypothesis that in any open system, communications crossing domain boundaries cannot be protected from, intentional or accidental, manipulations. The behaviour itself of a component creates a problem of trust, in its being compliant with the role that the component is supposed to play within the system.

  To protect sensible data and resources, as well as to authenticate the identity of components, many techniques, generally relying on a “secure” cryptographic system, have been developed. Nonetheless, validating the safeness of an engineered cryptographic system is still an open problem that cannot be ignored in the general context of open system modeling.

- **SYSTEM ANALYSIS.** The design and verification of open systems clearly result in being more complex than usual system design and verification. Not only the dynamics and partial accessibility make the analysis more complex, but many of the properties and techniques, which have been extensively studied and settled, need to be rethought in order to embrace the features of open systems. For instance, the traditional concepts of successful termination and absence of deadlocks, that have a clear meaning for a usual concurrent system, fall short of an open system that may not be fully specified, or whose full specification may not be available at a given instant of the system’s life. Moreover, verification techniques must often be suitable to be dynamically operated, so, for instance, as to be able to re-check a system every time that its structure changes.

As witnessed also by the above mentioned IST Research Programme, the area of models, languages and techniques for open systems is a very lively research area.
Fostered by the latest technological developments and with both practical and theoretical implications, it actually embraces several research fields addressing the issues discussed above in different contexts and with partially different aims, but often the essence of the problems is the same, or at least comparable.

The open system metaphor, that we informally discussed above, applies, either in full or in part, to many issues of interest for these different fields, and can help in uniformly approaching them. In particular, Chapter 2 surveys some of the proposals in the literature, which address problems that can be brought back to open system modeling. The different problems addressed and the solution proposed are presented with respect to the issues mentioned above.

1.2 Contributions of the thesis

In this thesis we have investigated the issue of modeling and verifying the aspects relative to the interaction of components in open systems. We have addressed the following topics:

1. a model for component interaction within open systems,

2. an extension of the model for component adaptation,

3. an extension of the model for system security.

1.2.1 A model for component interaction

In accordance with the vast majority of the ongoing research on the topic, we address interaction by equipping traditional component interfaces with behavioural descriptions, separating the observable behaviour of a component from its internal computation and implementation. Behavioural interfaces declare the behaviour that components autonomously offer to their environment in terms of communication actions of a suitable $\pi$-calculus based process algebra, the IP-calculus, that we have defined. Behavioural interfaces also address heterogeneity, as such they are an abstraction of the behaviour of a component independent of its actual implementation.

Arguing that dynamics and incomplete accessibility of an open system prevent a complete analysis of global properties, we focus on the verification of local properties, where local is read as regarding a finite extension in time. The behaviour of each single component is partitioned in finite interaction patterns. Interaction patterns consist of a behavioural expression, written in a non-recursive fragment of $\pi$-calculus, and an explicit indication of the references (names) the component offers to be connected to the other components in the environment.

Due to its expressiveness as far as dynamics and communication network reconfiguration are concerned, $\pi$-calculus has been chosen. The calculus is based on
the notion of names as abstractions enabling communications, that may be dynamically shared among components. Usually, names are interpreted as communication channels, but, as shown in the following, they can be used to model the more general concept of reference (to a channel, a resource, a capability). While many other proposals are based on Finite States Machines for describing behaviours, because of the computational difficulty of dynamically verifying a full-fledged \(\pi\)-calculus based system description, see for instance [CFTV99], the local analysis we adopted permits a reduction of such difficulties, while allowing for much of the expressiveness of the calculus.

Moreover, we have equipped the calculus with an explicit binding mechanism, according to the previously discussed needs for dynamic binding that open systems present. In general, it is not reasonable to assume that a component joining a system is aware of the channels or resources available in the system. Each interaction pattern is hence provided with a list of the references, called open variables, that it offers to be dynamically connected with its environment.

A system is modeled as a session in which interaction patterns are connected together and interact with each other. The connection of an interaction pattern to a session is regulated by a join operation explicitly connecting open variables of the joining interaction pattern to those of the interaction patterns already in the system.

The assumptions made facilitate the definition of a suitable notion of correctness, and its verification, for a session. Thanks to the finiteness of the information contained in a session about the behaviour of its patterns, it is possible to effectively verify the absence of critical errors within the so-far specified session, even if part of its overall behaviour is not yet known. Verification is based on finite model checking techniques (see Section 1.5.3 for a brief introduction), which are well-settled and widely used techniques. This (partial) correctness notion has been formalised in terms of the acceptability of an open session, that consists in the potential existence of an interaction pattern, called completion, that joins the session and allows all the interaction patterns that constitute the session to reach a successful termination.

While it is not possible to make statements about the overall potential interaction that will occur within a session, and that may be only partially specified at a given instant, acceptability guarantees the absence of unrecoverable interaction failures in the session “as is” at the verification instant.

In this sense, acceptability allows for the dynamical coordination of interaction patterns by forbidding the access of harmful patterns for the session. It follows that acceptability can be guaranteed throughout the life of a session, possibly indefinitely joined by correct finite interaction patterns, which may hence accomplish their tasks in a properly assembled system. Informally speaking, an interaction pattern that is admitted to a session commits itself to an interactive transaction within the session which is verified to be error free.

The model developed naturally applies also to the verification of standard closed system, since they can be seen as a particular case of open systems with limited
dynamics (for instance, acceptability of a closed session reverts to standard deadlock freedom).

Finally, acceptability checking is supported by a completion construction algorithm, that is proved to always be terminating and correct.

1.2.2 A model for component adaptation

According to the notion of acceptability, coordination of components in an open system requires a quite stringent matching of component behaviours. Either an interaction pattern fulfills acceptability, or it is prevented from accessing a session.

A complementary approach consists in developing techniques aimed at adapting components that are unable to interact with each other “as they are”, but that could be made interoperable, i.e. they could be enabled to interact without spoiling the correctness of a session, by an appropriate adaptation of their behaviours.

Component adaptation has been addressed by several studies in the literature, and in particular in the field of Component-Based Software Engineering (CBSE), [Cam99, Hei99, GS01], where the possibility for application builders to easily adapt off-the-shelf software components to work properly within their application is a must for the creation of a true component marketplace and for component deployment in general [BW98]. Adaptation generally consists in the deployment of a third component, an adaptor, which facilitates, by mediating, the interaction of components that have incompatible behaviours.

The open system verification framework we developed, can be easily extended to deal with the component adaptation problem, by interpreting an adaptor as a particular completion, which makes the session consisting of the two mismatching components correct.

Other than contributing to the ongoing research about component adaptation, the interest of this extension is also in validating the expressiveness of the interaction pattern framework. The results we obtained are:

- The development of a high-level language for declaratively expressing the required adaptation among the two components, called mapping. Differently from other proposal, where the adaptor designer is forced to specify the actual behaviour of the adaptor, the adoption of a high-level, abstract language permits the mapping designer to concentrate on the issues of adaptation, which requires human intelligent support, while delegating the actual adaptor construction process to an automatic procedure.

- The development of an automatic adaptor construction procedure. Exploiting the reading of an adaptor as a particular case of completion, the mapping abstract specification is used to automatically devise an adaptor by means of a variant of the completion construction algorithm. Indeed, in essence, adaptation is the process of making a session correct.
• The use of a name based formalism, like the IP-calculus, instead of the more widely adopted Finite State Machines, for expressing adapter’s behaviour. This allows an enhanced expressiveness, since it is possible to devise adapters able to deal with some forms of network reconfiguration, in a framework that, due to its finiteness hypothesis, does not appear less effective than the other proposals.

1.2.3 A model for system security

Finally, we have investigated the applicability of the IP-calculus to security issues, which, as explained, are a significant aspect of interaction in open systems. Security properties are enforced by security protocols, which rely on a cryptographic system in order to enforce the desired properties. The formal analysis and verification of security protocols is motivated by the fact that, even assuming the proper functioning of the cryptographic system (i.e. secrets cannot be decrypted), formalising the intended behaviour of a protocol and verifying that it actually enforces the desired properties is far from trivial. The formal characterisation of the properties themselves is still an open research problem.

Again, the application of the framework to a specific problem has the twofold aim of validating the expressiveness of the framework and contributing to the ongoing research in the field.

As far as the first point is concerned, the IP-calculus revealed particularly suitable for formalising security protocols, which typically consist in the interaction of a set of actors, called principals, with a finite behaviour. Moreover, exploiting the abstractness of name passing calculi, cryptographic keys are straightforwardly interpreted as names, while cryptographic primitives are embedded into communications, as the proper matching of a sent cryptogram and the corresponding decrypting keys owned by a receiver. These choices permit our language to model security protocols with minimal linguistic changes, obtaining a variant of the language called cIP-calculus (for cryptographic IP-calculus).

On the other hand, as far as verification is concerned, we have further developed and improved existing techniques, covering a reasonably complete spectrum of cases, including the non-trivial ones of symmetric and asymmetric keys and multi-session attacks, within a unique verification methodology. The developed verification methodology consists of three steps:

1. PROTOCOL FORMALISATION. Protocols are usually expressed in an ambiguous informal language by providing the behaviour of the participating principals. Principal behaviour needs hence to be formalised into interaction patterns of the cIP-calculus. This step requires the disambiguation of the informal language.

2. PROPERTY FORMALISATION. Formalising the security property that a protocol is supposed to guarantee is one of the crucial issues in all the approaches
to protocol verification present in the literature. We have extended the idea of Magic Instance, [AG99], according to which a protocol is secure if the data actually communicated are the same as those expected in a (magically) correct execution of the protocol. We have developed a logic which expresses statements about, not only the expected values of the protocol, but also the principals which communicate such data, and the knowledge that an intruder can acquire by interacting within the protocol. The set of properties that the logic permits to be expressed extends those of the Magic Instance, and includes secrecy, integrity and authentication. Moreover, the logic permits the issue of multi-session attacks to be naturally dealt with, by allowing for quantification over the multiple instances of a principal that may participate in more interleaved sessions of the protocol.

3. PROPERTY VERIFICATION. Protocol verification is carried out, similarly to acceptability verification and adaptor construction, by (model-) checking the validity of the formula throughout, namely at the termination points, the execution of a, possibly multi-session, run of the protocol, where the open environment plays the part of an intruder. Further strengthening the interpretation of a (multi-session) security protocol as an open system, the cIP-calculus semantics has been devised so as to possibly provide a model for a formula. The model, and hence the validity of the formula, depends on the communications which have occurred, the principals that have joined the protocol session, and the knowledge that has been disclosed to the environment.

A protocol is considered safe if the desired property holds in all the termination points. A run ending in a point where the property does not hold constitutes an attack on the protocol.

Technically, verification is based on a symbolic semantics we developed on top of the semantics of the cIP-calculus, that keeps the number of the possible states of a run of the protocol bounded. Otherwise, the infinite possibilities of attack that the intruder, according to universally accepted assumptions, is able to perform, would make the number of states infinite and the whole verification process unfeasible. Our symbolic semantics benefits from a strict correspondence among symbolic and concrete traces.

Valuable features of the cIP-calculus based verification are its completeness with respect to the kind of protocols and properties that can be analysed, the expressiveness and ease of use of the language and associated logic and the strict correspondence between symbolic and non symbolic semantics.

Finally, the study, formalisation and verification of security properties has given us insights into how to express more general properties about open systems, and suggestions on effective ways to dynamically enforce them. Some preliminary ideas have already been developed, but the main issues are scope for future work.
1.3 Outline of the Thesis

This introductory chapter ends by providing some technical background.

Chapter 2 surveys some of the related approaches, illustrating the context in which our research has been carried out.

In Chapter 3, IP-calculus, the calculus for finite interaction patterns, is introduced together with the notion of session. Standard correctness properties for closed, i.e. completely specified, sessions are discussed and acceptability for open sessions is introduced. The algorithmic verification of acceptability is then illustrated and proved correct.

In Chapter 4 the framework is extended with the adaptor construction methodology: the mapping language is illustrated and the algorithm for acceptability modified in order to produce adaptors.

Chapter 5 deals with security. The cIP-calculus, the cryptographic extension of IP-calculus is defined to model the principals of a security protocol, while sessions host multiple runs of the protocol. The logic for expressing security properties in terms of the outcome of the interaction happening in a session is defined. The analysis of the behaviour of a session is performed via symbolic analysis of traces.

Chapter 6 sums up the contribution of the thesis and sets out some directions for future work that have emerged from our research.

1.4 Origin of chapters

Part of the research presented in this thesis has been presented in published papers. Those works has been developed jointly with Antonio Brogi, Carlos Canal, Gianluigi Ferrari, Emilio Tuosto and Franco Turini.

The IP-calculus has been introduced in [BBT01a], [BBT01b].

Adaptors construction is discussed in [BBC02b], [BBC02a], [BBC02c], [BBC03].

Works regarding security and the verification of properties of open systems are [BBFT01], [BBFT02].
1.5 Technical background

1.5.1 Transition Systems

Transition systems are a standard way to give semantics to a computational, or more in general evolving system. The static configuration of the system is characterised by its internal state, i.e. a semantically interesting abstraction of it.

Evolutions of the system, which typically follow a set of rules, consist in changes of its state. A transition system is a mathematical model of a system and its evolutions.

**Definition 1.1 (Transition system)** A transition system is a couple $(\Gamma, \rightarrow)$, with $\Gamma$ a set of elements $\gamma$ called states, and $\rightarrow \subseteq \Gamma \times \Gamma$ a binary relation, called transition relation.

Hereafter, $\gamma \rightarrow \gamma'$ stands for $\langle \gamma, \gamma' \rangle \in \rightarrow$, where $\gamma$ and $\gamma'$ are called the source and, respectively, the target of the transition.

Sometimes, external effects of a transition may be semantically significant, and hence a set of observable actions, or labels, is associated to the transition system, yielding a labelled transition systems. Note how the label can represent the behaviour of the system by abstracting from its internal structure.

**Definition 1.2 (Labelled transition system)** A labelled transition system is a tuple $(\Gamma, A, \rightarrow)$ where $\Gamma$ is a set of states, $A$ is a set of labels and $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is the (labelled) transition relation.

Hereafter, $\gamma \xrightarrow{a} \gamma'$ stands for $\langle \gamma, a, \gamma' \rangle \in \rightarrow$, $\gamma \xrightarrow{a'} \gamma'$ for $\langle \gamma, a, \gamma' \rangle \notin \rightarrow$, $\gamma \rightarrow \gamma'$ for $\exists a \in A. \langle \gamma, a, \gamma' \rangle \in \rightarrow$, and $\gamma \xrightarrow{a} \rightarrow \gamma'$ for $\forall a \in A, \forall \gamma' \in \Gamma. \gamma \xrightarrow{a} \rightarrow \gamma'$.

**Definition 1.3 (Closures)** The transitive closure of the relation $\rightarrow$ is indicated by $\rightarrow^+$. $\gamma \xrightarrow{w^+} \gamma'$ stands for $\exists n > 0, \gamma_1, \ldots, \gamma_n, a_1, \ldots, a_n$ such that

$$\gamma \xrightarrow{a_1} \gamma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \gamma_n$$

and $\gamma' = \gamma_n, w = a_1, \ldots, a_n$. The transitive and reflexive closure, where $n \geq 0$ in the above definition, is indicated by $\rightarrow^*$.

A transition system can be represented through a graph, in which the states form the nodes and the arcs represent the possible transitions between them. The dynamic behaviour of a system defines the form of the graph, called transition graph. Some auxiliary definitions are needed. Hereafter, we assume as given a labelled transition system $(\Gamma, A, \rightarrow)$.

**Definition 1.4 (Derivative)** State $\gamma'$ is an immediate derivative of $\gamma$, if $\gamma \xrightarrow{a} \gamma'$. It is simply a derivative, if $\gamma \xrightarrow{w^*} \gamma'$. 

**Definition 1.5 (Reachable states)** The set of the reachable states of a state \( \gamma \) is
\[
ds(\gamma) = \{ \gamma' | \gamma \xrightarrow{w}^* \gamma' \}.
\]

Note that the union of the set of derivatives of the initial states of a system defines all reachable states.

We can now formalize the notion of derivation graph.

**Definition 1.6 (Derivation graph)** Given a state \( \gamma \) and its set of derivatives \( ds(\gamma) \), the derivation graph of \( \gamma \) is
\[
dg(\gamma) = \langle ds(\gamma), A, \{ (\gamma_i, a, \gamma_j) | \gamma_i \in ds(\gamma) \land \gamma_i \xrightarrow{a} \gamma_j \} \rangle
\]
where \( ds(\gamma) \) is the set of nodes, \( A \) is the labelling alphabet, and the third set defines the arcs.

The possible patterns of behaviour of a system are obtained by visiting its derivation graph. The sequences of consecutive transitions that describes the behaviour are called computations.

**Definition 1.7 (Computation (or trace))** Let \( \gamma_i \xrightarrow{a} \gamma_j \) be a transition. A computation of \( \gamma \) is a sequence of transitions \( \gamma = \gamma_0 \xrightarrow{a_0} \gamma_1 \xrightarrow{a_1} \ldots \) starting from \( \gamma_i \), and such that the target of any transition coincides with the source of the next one. We let \( \xi, \xi', \xi_1, \ldots \) range over computations, and we write \( \epsilon \) for the empty computation. With \( C(\gamma) \) we indicate the set of computations of \( \gamma \):
\[
C(\gamma) = \{ \xi_i | \xi_i = \gamma \xrightarrow{a_0} \gamma_1 \xrightarrow{a_1} \ldots \},
\]
while \( C(\gamma, \gamma') \) indicates the set of computations with source \( \gamma \) and target \( \gamma' \) (source and target are extended in the obvious way to computations):
\[
C(\gamma, \gamma') = \{ \xi_i | \xi_i = \gamma \xrightarrow{a_0} \gamma_1 \xrightarrow{a_1} \ldots \gamma' \}.
\]

Sometimes is useful to have a linearization of all computations that a system may engage in. A possibility is to get the unfolding of the derivation graph, thus yielding a tree of computations (as abstractly defined below, where summation represents sibling relation and \( \cdot \) descendant relation).

**Definition 1.8 (Derivation tree)** Given a state \( \gamma \) and its derivation graph \( dg(\gamma) \), the derivation tree of \( \gamma \) is defined inductively as follows
\[
dt(\gamma) = \sum_{(\gamma, a_i, \gamma_i) \in dg(\gamma)} a_i \cdot dt(\gamma_i)
\]
It is clear from the above definition that any path in the derivation tree of a state \( \gamma \) represents a computation starting at \( \gamma \).
1.5.2 The $\pi$-calculus

In this section we briefly recall the $\pi$-calculus, a model of concurrent communicating processes based on the notion of naming. We follow [MPW92]. This presentation is meant as a reference in order to compare the calculus we will define with its full-fledged ancestor.

**Definition 1.9 (Syntax)** Let $\mathcal{N}$ be a countable, infinite set of names ranged over by $a, b, \ldots, x, y, \ldots$, and let $\tau$ be a distinguished element such that $\mathcal{N} \cap \{\tau\} = \emptyset$. Processes (denoted by $P, Q, R, \ldots \in \mathcal{P}$) are built from names according to the syntax

$$P ::= \begin{align*}
0 & \quad \text{nil} \\
\pi.P & \quad \text{prefix} \\
P + P & \quad \text{summation} \\
P|P & \quad \text{parallel composition} \\
(\nu x)P & \quad \text{restriction} \\
[x = y]P & \quad \text{matching} \\
A(y_1, \ldots, y_k) & \quad \text{constant definition}
\end{align*}$$

$$\pi ::= \begin{align*}
\tau & \quad \text{silent prefix} \\
x(y) & \quad \text{input} \\
\overline{xy} & \quad \text{output}
\end{align*}$$

where $A$ is an agent identifier. (Alternatively, the last production for $P$ can be replaced by $!P$, replication). Hereafter, the trailing $0$ will be omitted (i.e. we will write $\pi$ instead of $\pi.0$).

The process $\pi.P$ can perform the atomic action $\pi$ and then evolve in accordance to $P$. The input prefix binds the name $y$ in the prefixed process. Intuitively, some name for $y$ is received along the link named $x$. The output prefix does not bind the name $y$ which is sent along $x$. The silent prefix $\tau$ denotes an action which is invisible to an external observer of the system. Summation denotes non-deterministic choice. The operator $|$ describes parallel composition of processes. The restriction operator $(\nu x)$ binds the name $x$ in the process $P$ that it prefixes, making $x$ a unique name local to $P$. Matching $[x = y]P$ acts as an if-then operator: (only) if $x = y$ the process evolves in accordance to $P$. Finally, each agent identifier $A$ has a unique defining equation of the form $A(y_1, \ldots, y_k) = P$, where the $y_i$'s are the only free names (see below) of $P$ and $y_i \neq y_j$ if $i \neq j$. Alternatively, $!P$ behaves like infinitely many copies of $P$ running in parallel.

The $\pi$-calculus is equipped with a labeled transition system semantics. Several transitions systems have been proposed in order to address different semantical aspects. They mainly differ in terms of the moment in which variables are bound to names. For instance, in the *early semantics* the bound input name (placeholder)
is instantiated in the axiom for the input action (see rule \textit{Ein} in Tab. 1.2) as soon as the action is performed, while in the \textit{late semantics} it is instantiated only when a communication occurs (see rule \textit{Com} in Tab. 1.1), and in the \textit{open semantics} it can be arbitrarily delayed (see, e.g., [Pis99] for a quick survey). We present here the \textit{late} and the \textit{early semantics}.

### Late semantics

The late operational semantics of the \(\pi\)-calculus is defined in Figure 1.1, in \textit{SOS} (Structural Operational Semantics) style. The labels of transitions, ranged over by \(\mu\), are \(\tau\) for silent actions, \(xy\) for input, \(\overline{xy}\) for free output, and \(x(y)\) for bound output.

The following table defines the notion of free names \(fn(\mu)\), bound names \(bn(\mu)\), and names \(n(\mu) = fn(\mu) \cup bn(\mu)\) of a label \(\mu\), and of the subject \(sbj\), and the object \(obj\) of input and output actions:

<table>
<thead>
<tr>
<th>Kind</th>
<th>(\mu)</th>
<th>(fn(\mu))</th>
<th>(bn(\mu))</th>
<th>(sbj(\mu))</th>
<th>(obj(\mu))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silent move</td>
<td>(\tau)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>Free output</td>
<td>(\overline{xy})</td>
<td>({x,y})</td>
<td>(\emptyset)</td>
<td>({x})</td>
<td>({y})</td>
</tr>
<tr>
<td>Input and bound output</td>
<td>(x(y), \overline{xy})</td>
<td>({x})</td>
<td>({y})</td>
<td>({x})</td>
<td>({y})</td>
</tr>
</tbody>
</table>

Functions \(fn\), \(bn\) and \(n\) are extended in the standard way to processes.

A substitution is a function \(\sigma : \mathcal{N} \to \mathcal{N}\) which is almost everywhere the identity. If \(x_i \sigma = y_i\) for all \(i\) with \(1 \leq i \leq n\) (and \(x \sigma = x\) for all other names \(x\)), we sometimes write \([y_1/x_1, \ldots, y_n/x_n]\) or \([\overline{y}/\overline{x}]\) for \(\sigma\). Then, \(P \sigma\) denotes the agent obtained from \(P\) by simultaneously substituting \(z \sigma\) for each free occurrence of \(z\) in \(P\) for each \(z\), with change of bound names to avoid captures. In particular the following two rules hold where \(\equiv\) denotes syntactic identity

\[
(x(y).P) \sigma \equiv x\sigma(y').P[y'/y]\sigma \quad \text{where} \quad y' \not\in fn((\nu y)P, P\sigma) \land y'\sigma = y'
\]

and

\[
(\nu y)P \equiv (\nu y')P[y'/y]\sigma \quad \text{where} \quad y' \not\in fn((\nu y)P, P\sigma) \land y'\sigma = y'.
\]

### Early semantics

In the \textit{early} version of the operational semantics, the placeholder of the input is instantiated in the axiom for early input (\textit{Ein}): \(x(y).P \xrightarrow{xy} P[w/y]\). Figure 1.2 shows the \textit{early} transition system of the \(\pi\)-calculus.

Here the labels of transitions are \(\tau\) for silent actions, \(xy\) for free input, \(\overline{xy}\) for free output, and \(\overline{x(y)}\) for bound output.

<table>
<thead>
<tr>
<th>Kind</th>
<th>(\mu)</th>
<th>(fn(\mu))</th>
<th>(bn(\mu))</th>
<th>(sbj(\mu))</th>
<th>(obj(\mu))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silent move</td>
<td>(\tau)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>Free input and output</td>
<td>(xy, \overline{xy})</td>
<td>({x,y})</td>
<td>(\emptyset)</td>
<td>({x})</td>
<td>({y})</td>
</tr>
<tr>
<td>Bound output</td>
<td>(\overline{x(y)})</td>
<td>({x})</td>
<td>({y})</td>
<td>({x})</td>
<td>({y})</td>
</tr>
</tbody>
</table>
1.5. TECHNICAL BACKGROUND

act : $\mu.P \xrightarrow{\mu} P$

match : $P \xrightarrow{\mu} P'$

$[x = x]P \xrightarrow{\mu} P'$

par : $P \xrightarrow{\mu} P'$

$P \xrightarrow{\mu} P$, $bn(\mu) \cap fn(Q) = \emptyset$

$P \xrightarrow{\mu} P'$

sum : $P \xrightarrow{\mu} P'$

$P + Q \xrightarrow{\mu} P'$

res : $(\nu x)P \xrightarrow{\mu} (\nu x)P'$, $x \notin n(\mu)$

open : $(\nu y)P \xrightarrow{\mu} P'$, $y \neq x$

close : $P \xrightarrow{\tau} (\nu y)(P'|Q')$

$P \xrightarrow{\tau} Q \xrightarrow{x(y)} Q'$

com : $P \xrightarrow{\tau} P'$

$P \xrightarrow{\tau} P' | Q \xrightarrow{x(y)} Q'$

$P \xrightarrow{\tau} P' | Q'[y/z]$

def : $P[y_1/x_1, \ldots, y_n/x_n] \xrightarrow{\mu} P'$

$A(y_1, \ldots, y_k) \xrightarrow{\mu} P'$, $A(x_1, \ldots, x_n) \overset{def}{=} P$

Figure 1.1: Late transition system for the $\pi$-calculus.

Structural Congruence

The structural congruence $\equiv_P$ on processes\(^1\) is defined as the least congruence satisfying the following clauses:

- $P \equiv_P Q$ if $P$ and $Q$ are $\alpha$-equivalent (in symbols $P =_\alpha Q$); to be more precise:
  
  $(\nu x)P \equiv_P (\nu y)(P[y/x])$ if $y \notin fn((\nu x)P)$, and
  
  $x(y)P \equiv_P x(z)(P[z/y])$ if $z \notin fn(x(y)P)$,

- $(P/\equiv_P, +, 0)$ and $(P/\equiv_P, |, 0)$ are commutative monoids,

- $(\nu x)(\nu y)P \equiv_P (\nu y)(\nu x)P$, $(\nu x)P \equiv_P P$ if $x \notin fn(P)$,
  
  $(\nu x)(P | Q) \equiv_P (\nu x)P | Q$ if $x \notin fn(Q)$.

We call a variant of $P \xrightarrow{\mu} Q$, a transition which only differs in that $P$ and $Q$ have been replaced by structurally congruent processes, and $\mu$ has been $\alpha$-converted (under the assumption that a name bound in $\mu$ includes $Q$ in its scope).

\(^1\)We will omit $\mu$, whenever clear from the context.
Figure 1.2: Early transition system for the \( \pi \)-calculus.

The transition in the conclusion of each semantic rule, or axiom, represents all its variants. This is usually handled by implicitly assuming the rule

\[
P' \equiv_p P \quad \frac{P \xrightarrow{\mu} Q \equiv_p Q'}{P' \xrightarrow{\mu} Q'}
\]

Note how this rule permits some semantic transition to be dealt with by means of congruence rules. For instance, rules match and def (or, alternatively, the one for the \(!\) operator) can be made no longer necessary by assuming

- \([x = x]P \equiv_p P\),
- \(A(a_1, \ldots, a_k) \equiv_p P[a_1/y_1, \ldots, a_k/y_k]\), if \(A(y_1, \ldots, y_k) = P\) (or, alternatively, \(!P \equiv_p P[!P]\)

The following proposition shows how late and early semantics are related. The proof proceeds by induction on the rules of early and late operational semantics.

**Proposition 1.1** Let \( \rightarrow_E \) and \( \rightarrow_L \) denote the early and the late transition relation. Then,
Equivalences

The operational semantics of processes is sometimes too intensional and detailed to study the behaviour of distributed concurrent systems. Therefore behavioural equivalences based on the notion of bisimulation are introduced. As usual, there is a strong and a weak version of bisimulation. The former treats all actions uniformly, while the latter abstracts from invisible or silent actions because it discards them. We only present the strong version of bisimulations (weak bisimulations are obtained by suitably, according to the chosen semantics, abstracting silent actions, see, e.g., [Pis99] for a quick overview).

The classical definition of bisimulations for π-calculus are the early and late ones. As for the operational semantics, the two notions differ for the moment in which instantiation of names occurs (i.e., input for the early case and communication for the late one). We start with the definition of the late equivalence.

Definition 1.10 (Late bisimulation) A binary relation \( S \) on processes is a late simulation if \( P S Q \) implies that

- If \( P \xrightarrow{\mu} P' \) and \( \mu \) is \( \tau \), \( \overline{xz} \) or \( \overline{t(y)} \) with \( y \notin \text{fn}(P,Q) \), then for some \( Q' \), \( Q \xrightarrow{\mu} Q' \) and \( P' S Q' \)
- If \( P \xrightarrow{x(y)} P' \) and \( y \notin \text{fn}(P,Q) \), then for some \( Q' \), \( Q \xrightarrow{x(y)} Q' \) and for all \( w \), \( P'[w/y] S Q'[w/y] \)

The relation \( S \) is a late bisimulation if both \( S \) and \( S^{-1} \) are late simulations. \( P \) is late bisimilar to \( Q \) (written \( P \approx_L Q \)) if there exists a late bisimulation \( S \) such that \( P S Q \).

Note that late simulations have a strong requirement on bound input actions.

Definition 1.11 (Early bisimulation) A binary relation \( S \) on processes is an early simulation if \( P S Q \) implies that

- If \( P \xrightarrow{\mu} P' \) and \( \mu \) is any action with \( \text{bn}(\mu) \cap \text{fn}(P,Q) = \emptyset \), then for some \( Q' \), \( Q \xrightarrow{\mu} Q' \) and \( P' S Q' \)

The relation \( S \) is an early bisimulation if both \( S \) and \( S^{-1} \) are early simulations. \( P \) is early bisimilar to \( Q \) (written \( P \approx_E Q \)) if there exists an early bisimulation \( S \) such that \( P S Q \).

1. \( P \xrightarrow{\pi y} E \) \( P' \) \( \iff \) \( P \xrightarrow{\pi y} L \) \( P' \)
2. \( P \xrightarrow{\pi(y)} E \) \( P' \) \( \iff \) \( P \xrightarrow{\pi(y)} L \) \( P' \)
3. \( P \xrightarrow{x(y)} E \) \( P' \) \( \iff \) \( \exists P'', w . P \xrightarrow{x(w)} L P'' \land P' \equiv P''[y/w] \)
4. \( P \xrightarrow{\tau} E \) \( P' \) \( \iff \) \( P \xrightarrow{\tau} L \) \( P' \)
1.5.3 Model checking for verification

The verification techniques developed in this thesis fall in the class of model checking. Since this thesis is not, anyway, about model checking, we simply recall the basic principles of this verification technique, referring to the literature for a deeper introduction, [CW+96].

Model checking, together with theorem proving, is a well established verification technique that relies on building a finite model of a system and checking that a desired property holds in that model. Generally, the checking is done by exhaustive state space search whose termination is guaranteed by the finiteness of the model. The issue is hence finding efficient ways of exploring large state spaces.

Two are the main approaches to model checking: temporal model checking – the property is expressed in a temporal logic [Pnu81] and the model of the system as a finite transition system –, and automaton model checking – both the property and the system are expressed as automata whose behaviours are checked for conformance–.

Advantages of model checking are that it is generally completely automatic, it can provide counterexamples and check partial specifications. Its drawback is the state explosion problem, even if, by means of efficient procedures, systems with $10^{120}$ reachable states have been checked, [CW+96]. Another promising direction is the use of abstraction techniques in order to deal with unboundedness in the number of reachable states.

In Chapter 5 we have developed an automatic, abstract (symbolic) model checker for the verification of security protocols, which is based on an ad hoc logic and whose returned counterexample, if there is one, is an attacker of the protocol.
Chapter 2

State of the art

Contents

2.1 Coordination models and languages ............................................. 20
  2.1.1 Motivations ................................................................. 20
  2.1.2 Coordination models ..................................................... 21
  2.1.3 Discussion .................................................................. 24

2.2 Software Architecture ............................................................... 25
  2.2.1 Motivations .................................................................. 25
  2.2.2 Architecture Description Languages ................................. 26
  2.2.3 Future perspectives ....................................................... 29
  2.2.4 Discussion .................................................................. 30

2.3 Component Based Software Engineering ................................. 31
  2.3.1 Motivations .................................................................. 31
  2.3.2 Current trends ............................................................. 32
  2.3.3 Future perspectives ....................................................... 33

As it was already widely recognised a decade ago “As the size of software increases, the algorithms and the data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organisation of the overall system [...] presents a new set of design problems”, [GS93].

As illustrated in the Introduction by means of the open system metaphor, the current software development is made even more complex as far as the aspects of the organisation of the overall system are concerned, mainly because the overall system structure may be open.

In order to cope with this increased complexity, regarding different aspects of system design, several research communities have addressed the problems from different perspectives, with different methodologies and goals, proposing partially overlapping
solutions, that the novelty of the field and, perhaps, the rapid pace of technology evolution, have not yet allowed to be uniformly settled.

Complexity is dealt with by raising the level of abstraction, and in the shift of software design from the usual process of developing requirements into code, towards assembling components into open systems, almost all the approaches pay attention to composition and interaction issues.

Among the work carried out by the communities which address composition and interaction, the approaches of COORDINATION, SOFTWARE ARCHITECTURE and COMPONENT BASED SOFTWARE ENGINEERING offer the most relevant contributions for our thesis concerning the behavioural aspects of software composition.

All of them share the common aim of describing and reasoning about systems assembled from components, witnessing the new challenges brought about by the openness of component-based systems. Even if quite recent, they attract the interest of a mature research field, with an increasing number of conferences, tracks, publications, and industrial applications. Actually, many results are relevant for all three of the approaches, crossing boundaries that are rather fuzzy. For instance, the same model may be listed within different fields by different surveys. In what follows we recall some of the principal proposals in the huge production of the three fields, being aware that the following classification has mainly been adopted for ease of presentation.

2.1 Coordination models and languages

2.1.1 Motivations

Back in 1994, COORDINATION was proposed as an interdisciplinary field involving contributions from Economics, Organization Theory, Computer Science, Biology, Linguistics, and others, since each of these fields addresses, under a different perspective, the “managing of dependencies between activities”, [MC94]. Moreover, it has also been claimed that a theory of behavioural coordination of independent actors has been needed. More specifically to Programming Languages, “Coordination is the process of building programs by gluing together active pieces”, [CG92], and it was initially motivated to cope with the increasing complexity of concurrent and distributed programming.

The key starting point for the development of a coordination theory, is that interaction concerns are orthogonal to computation concerns, and as such they must be described separately in system models, [CG92, PA98, ACG86, SBMW99]. This is still in contrast to what happens to most of the current technology infrastructures [CD99], where interaction is often “hard-wired” into the code of applications.

This separation is made even more evident by other authors, who call for the definition of a new computational model, claiming that the Turing Machine is not suitable for properly modeling interaction concerns, [Weg98], since it does not pro-
vide suitable abstractions to model the dynamical interplay between a computational entity and its environment. The new model, called *Interaction machine*, features reactive capabilities against an input stream, together with standard computational capabilities (the model is interpreted in terms of co-inductive algebras, whose potentially infinite behaviour is externally observed, one action at a time, while unfolding, in the style of standard bisimulation (Section 1.5)).

Accordingly, a further definition of coordination sounds like “[Coordination consists of …] The study of the dynamic topologies of interaction among Interaction Machines — [Weg98] —, and the construction of protocols to realize such topologies that ensure well-behavedness”, [Arb98]. This last definition motivates coordination models also as a means of verifying the well-behavedness of a system.

Another advantage of a clear separation of interaction from computation is that it typically implies that the coordination model results in being language (paradigm) independent, so that, other than dynamics, heterogeneity can be more easily addressed.

A coordination model, hence, consists of an abstraction of the autonomous entities which populate a system and the means by which they interact, while a coordination language “implements” the coordination model in terms of a specific linguistic paradigm.

### 2.1.2 Coordination models

Coordination models and their relative languages are typically classified into *Data driven* and *Control driven* models (languages), [PA98].

In the former, coordination explicitly depends on application data: coordination is pursued by means of a shared data space, where components share information concerning their interactive tasks. The coordination data space is accessed by means of a common language that abstracts from internal implementations. Since, in general, components are loosely coupled to each other in their dependence on only the common data space, the treatment of heterogeneity and dynamics is facilitated. Unfortunately, being the coordination code spread inside computation code (on whose data it depends), the lack of a proper “coordination layer” makes a proper separation of concerns often unfeasible.

On the other hand, control driven models are concerned with the control of components, in terms of how they communicate to each other, and how they react to external events. Components are seen as black boxes, accessible via a coordination interface, which typically presents the way in which a component can interact with its environment. Data handled by the computational processes are of scarce concern for the coordination framework which evolves because of state changes in processes and process topology, and the events that possibly occur within the environment. In this model the communication network topology, its possible dynamical reconfiguration, the events that allow it to evolve, and the kind of connections allowed are central issues. Computation and interaction concerns can hence be clearly distinct and the
verification of the system coordination properties can be more straightforward.

In the following we illustrate a few of the representatives of each class. This is not intended as a complete survey of the huge set of proposed coordination models, but as a recap of the addressed problems and approaches developed.

*Linda*, [ACG86, CG89], with all its many extensions, is one of the most successful and influential data driven coordination languages. It extends, potentially any programming paradigm with coordination capabilities in a straightforward and natural way. It is based on a shared data space, the *tuple space*, where each component can read or write information in form of tuples. Tuples are typed and they can represent active processes. They are managed by primitives like \texttt{in(t)}, \texttt{out(t)}, \texttt{eval(t)}, that retrieve, put or execute a tuple in the tuple space. Primitives act on tuples by means of *associative pattern matching*. Components are loosely coupled since communication is basically anonymous and asynchronous.

The expressiveness and simplicity decreed the success of the basic language which has been extended in various directions. The tuple space has been structured in a hierarchy, allowing for processes crossing the boundaries of nested tuple spaces (with mechanisms a l’a Ambient Calculus, [CG00], [CGZ95]). The tuples managing mechanism has been extended with constraining rules, [ML95]. In *Objective Linda*, [Kie96], objects share hierarchies of object spaces, where objects can exchange references. The expressiveness of the language primitives has motivated several comparisons, [BJ99, BGZ00], while distributed programming environments have been designed in the light of the Linda concepts (e.g. Jada [CR96], Jini and JavaSapace [SUNc] technology).

*Shared Prolog*, [BC91]. This language is based on a black board shared by distributed Prolog programs. The Linda-like pattern matching is replaced by *logical unification* and coordination control is distributed among the processes by means of guarded clauses that synchronize the access to the common data structure. It represents an extension of the Linda paradigm, implemented for the case of a specific language.

*Klaim*, [DFP98]. This language extends the coordination principle on which Linda is based with *localities* allowing for process coordination in distributed and located tuple spaces. Processes move through localities and can refer to them as first order objects. Each process is hence aware of the location in which computation is performed. Moreover, mechanisms allowing for static and dynamic bindings are provided, so that each process can refer to its own environment as well as to the external environment it is visiting. The language is also provided with a type system that can be exploited to enforce security properties by means of access control policies, [DFP99], hence improving the scope of coordination.

*Multisets rewriting approaches*. These approaches, also based on shared data spaces, exploit the similarity of coordinated interaction with chemical reactions. The idea is that data are concurrently manipulated by conditional transformation rules, as when in a chemical reaction molecules react independently from each other.
The model abstracts from the programming paradigms used to define the data transformation rules, while it implements coordination by managing the possible concurrent applications of transformation rules. The basic paradigm is extended, adding structure to the "flat reactive chemical solution", and increasing the control of the applications on the transformation rules, allowing for a more expressive coordination of components.

The paradigm, has been used as a coordination model, the GAMMA model [BM96], but it has also been advocated as expressive Architecture Description Language, in the context of SOFTWARE ARCHITECTURE, that is, the CHAM formalism, [IW95], one of the many examples of the overlapping of the two fields.

Synchronizers, [FA94], are based on the Actor model, [Agh86], and express coordination patterns within a multi-object framework by means of constraints that restrict invocations to objects. Constraints are defined in terms of object interfaces, guaranteeing the language independence of the model. Constraints can express conditions on the data.

Manifold, [AHS93], is a control-driven coordination language, based on the IWIM model, [Arb96]. The model describes a system as a set of workers (the computational components) and a set of managers (the coordinators). Workers communicate through channels that are dynamically managed by the managers. Computational issues are hidden within workers, as well as language or paradigm dependencies. Workers are loosely coupled: they communicate through (local) ports, without needing any information about their partners. The managers are uniquely responsible for orchestrating the jobs of the workers, by connecting their ports by means of channels of a suitable type. Channel types, chosen in order to represent a minimal expressive set, can be composed in order to manage arbitrarily complex patterns of interactive communication.

More specifically, Manifold coordinators communicate via input/output ports connected by streams. The system changes its state due to the occurrence of events. Typically a Manifold coordinator consists of a series of networks of coordinated processes interconnected by streams. Processes are instantiated with actual components, by means of a form of late binding. The input and output of processes are redirected according to the stream topology. The occurrence of events, generated by component interaction, activates different networks, allowing for a form of dynamic reconfiguration. Events do not carry data, but only trigger a state change.

Coordinators can be structured, so that a coordinator can be seen by the other processes as a normal process, and in turn coordinated by an external coordinator, building a hierarchy of coordination protocols.

A further enhancement in this approach of interpreting coordination in terms of the properties and behaviour of the communication network, is $P_{\omega}$ [AM02], a channel based coordination language, in which the attention is completely focused on the construction of connectors by composing channels into coordinators. Structure and composition of channels, which may also feature mobility, impose a coordination
pattern to the connected components. Component interfaces lose their significance, except for indicating how components are connected to channels. Different “connection patterns” guarantee different properties about component interaction, allowing for system analysis, while an intuitive graphical representation of channels allows for visual coordination programming.

2.1.3 Discussion

The coordination community addresses the study of software composition, mainly focusing on the interaction between components and the more suitable abstractions to model it. Even if architectural issues are not a main concern, the approaches proposed by the coordination community often find application within more architectural fields, software architecture and component based software engineering in particular, and are, more in general, required by any approach to build the “large scale, highly reliable, evolving, always available systems” of the Network-Centric age, [Wal02].

Coordination models cover a lot of the features of open systems.

By the use of interfaces, in the case of control-driven models, or loosely coupled components, in anonymous communications through shared data spaces, coordination models address heterogeneity by abstraction in a quite successful way.

As well as for autonomy, which, in the end, is one of the motivating issues for coordination, they succeed in modeling and analysing the interactive emerging behaviour of component-based systems. Sometimes, the focus on the essence of interaction is so strong that components reduce only to the patterns of interaction they exhibit.

Loosely coupled data spaces address also some form of incomplete accessibility of the environment, as components do not need to have a complete knowledge of the other components that access the data space. This is also obtained, in control-driven models, e.g. [AM02], by providing some forms of dynamic binding of communication media (typically, channel names). Anyway, in general, the dynamical connection (and disconnection) of a component to a running open system is not adequately taken into consideration, nor is it supported by suitable linguistic abstractions, and dynamics is mainly understood as network reconfiguration.

Analogously, system verification techniques are often not suitably expressive to cope with the openness of systems, requiring a complete knowledge of the system under analysis, and being based on properties which are standard for closed systems. In general, the difficulty of lifting to the open system case the techniques, which have traditionally been applied to closed systems, has been recognised by several different studies (see, for instance, [BBB02] about lifting semantical equivalences of process calculi).

The coordination approach, hence, in its abstractly addressing interaction, appears as a valuable and promising contribution to the understanding of open systems and as a reference for the development of more practical, architectural
methodologies. However, further work is required to satisfactorily address all the issues raised by the new computational model.

2.2 Software Architecture

2.2.1 Motivations

More than a decade ago, the needs of promoting complex software system development to a properly mature engineering discipline motivated the birth of Software Architecture as a sub-field of software engineering. As the complexity of systems was increasing, the standard “code-oriented” methodologies appeared not suitable for sustaining the technology evolution. The benefit of a convergence of Software Engineering and (architectural) Programming Languages has extensively been recognised, [GM96].

According to one of the many definitions of Software Architecture as “The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time”, [GP95], a large effort has been devoted to defining the constitutive elements of software systems and providing suitable linguistic abstractions for expressing components and their relationships.

The main aim of this effort has been to settle system development as an architectural process, within a well-understood theory on the structure of the systems, the components and their interrelationships, so as to allow for design by patterns, verification of system structures and reuse of previously developed (sub-) systems.

This kind of research has been inspired by modularity and abstraction, which have been well understood at the code-level by the object oriented approach. The idea of enhancing objects with an explicit control over their communications in a distributed environment dates back to the late 70s with the introduction of the Actors model, [Hew97, Agh86, AH87], as a first step towards composition of autonomous objects.

After the work about the description and the analysis of recurrent design patterns in object-oriented code development, [GHJV94], the idea of exploiting recurrent design structures at an architectural level has also been adopted in Software Architecture, [GP95, Gar95, SDK+95, SG96]. The objective is to identify basic components and their interconnections and to study the patterns that more often recur in building an architecture out of components and interconnections. Clearly identifying the most suitable patterns in the early steps of design improves the quality of the design process. In this way, system design can benefit from enhanced system analysis techniques, based on a well understood set of architectural structures.

Some examples of recurrent architectural patterns are, according to [GP95], client server models, pipelines, filters, layered structures and blackboard systems.
Other than with design, software architecture is concerned with reuse, as they provide modularity and hence substitutability and refinement, with verification. Indeed, architectural descriptions can often be equipped with automatic verification tools (typically based on model checking techniques). Moreover also maintenance is improved, as modularity can be exploited for reconfiguration.

After about ten years, significant progress has been made. Mainly,

- the needs for an engineering approach has been fully recognised, both as a research area and as an industrial practice,

- a set of formal notation and tools for describing and analysing architectural designs, called Architecture Description Languages (ADL), have been proposed, some of which are illustrated in Section 2.2.2.

However, they claim that, despite the progress made, the field is still not a completely mature discipline [Gar00], and that the ongoing advent of open systems reveals new problems to be solved, [Sha99].

Moreover, some of the proposed ADL substantially differ from each other in terms of the issues they address, like, e.g., domain specific architectural features, architectural styles, event-based or distributed message passing communication protocols, support for the system analysis and simulation, architectural refinement, support for different kinds of connectors. Such disparities call for standards, expressed as theoretical models, e.g. the ACME model [GMW97] allowing for ADL comparison, as taxonomies, e.g. [MT97] and, more practically, as reading of ADL concepts by means of commonly used formalisms, like the UML specification language, [MR99].

2.2.2 Architecture Description Languages

Some ADLs promote an explicit distinction between components and connectors, arguing that such a distinction is more natural for architecture design. Intuitively speaking, the empirical practice of boxes and lines for describing a system highlights the conceptual differences between connectors and components. A model which explicitly maintains this difference, also by means of separate semantics and verification tools, more strictly adheres to such an intuition, facilitating the practical usage of the description formalism. Moreover, decoupling components from connectors permits the same component to be used with different, type-compatible, connectors, and vice-versa.

On the other hand, other proposals support the view of connectors as computational entities, hence components, in name of the minimality of the set of abstractions needed by an ADL to describe the system.

The differences between the two approaches are mainly a matter of expressiveness and must be evaluated in accordance with the context in which they are employed, e.g. foundational reasoning about the principles of software composition, or practical
application to the product line. Both the approaches, anyway, are clearly concerned with defining the abstractions which allow for a clear description and verification of component interaction in a better way. In what follows, some ADL are illustrated.

**UniCon** [SDK95] is based on an explicit distinction between components and connectors. The former can be primitive “computational entities” or composite components. A binary executable code is a primitive component, components can be hierarchically composed in new (composite) components. An abstraction for a pipeline or a remote procedure call are kinds of connectors. Systems are built by interconnecting (the interfaces of) components and connectors that must match the respective types. A standard set of connectors is provided in order to assembly systems, possibly according to some architectural pattern. UniCon allows for type-checking the composition of components and connectors, and, permits external verification tools to be interfaced with it during the development phase.

**Darwin** [MK96], a language originating from distributed software development, can express some forms of dynamic reconfiguration. It is presented as a declarative language in the sense that systems are described exclusively in terms of the services each component offers or requires. The language describes how more complex components, or systems, can be constructed by correctly connecting the appropriate services. Semantics of connections is given as $\pi$-calculus terms responsible of appropriately sharing references between services, so as to model dynamic reconfiguration. Anyway, in name of a strong commitment to declarativity, operational concerns, as well as behavioural analysis, are not a main focus of the original formulation of the ADL. On the other hand, in the subsequent paper [MKG99], each component has been enriched with a separate description of its behaviour by means of a finite state process algebra, whose actions are (required or offered) services, composed sequentially or as possible alternatives. Since states are finite (and new actions cannot be generated), recursion, which is admitted, reads as repetition of a behavioural pattern. Verification is performed by means of an exhaustive exploration of the state space, searching for states explicitly marked as “errors” or for deadlocked states.

**Wright** is an Architecture Description Language which embodies the approach of [AG97], regarding the formalisation of architectural connections. It maintains the explicit distinction between computational components and architectural connectors. Component behaviour is partitioned in ports. Connectors consist of a set of roles that components play in the connections, and in a gluing behaviour that components must respect when their ports are connected to roles. Again, behaviour is described by means of a finite state process algebra (based on CSP, [Hoa78]). Architectures are verified against compatibility of ports, roles and glue. A relation of refinement over behaviours is defined. If a behaviour is replaced by one of its refinements, correctness of the composition is preserved. Moreover, a trace specification language permits the designers to verify more general properties about computation. As the authors underline, the model as is, especially for the choice of CSP as an underlying model, does not support the dynamic evolution of the system.
Leda [Vel00, CPT99a] is an ADL, formally grounded on the basis of a process algebra, which extends the notions of compatibility, inheritance and polymorphism, widely accepted in the object-oriented paradigm, to the analysis of component behaviour. The behaviour of a single component is partitioned into facets, called roles, and composition and verification regard the point-wise compatibility of attached roles of different components. Differently from Wright, Unicon, and others ADLs, Leda is not equipped with an explicit notion of connector. The authors argue that i) components and connectors are conceptually equivalent autonomous entities, ii) the introduction of connectors as separate entities makes the analysis of properties, like compatibility and inheritance, unnecessarily more complex, and iii) the composition of components and connectors, in case of structured architectures, gives raise to hybrid elements with an unclear semantics. Behavioural analysis is supported by an underlying $\pi$-calculus based model. Inheritance and polymorphism are used to define the notion of behavioural compatibility, so as to allow for substituting a component with a component exhibiting a similar behaviour, pursuing reuse of components. Moreover, compatibility is also used for checking the correctness of the attachments between roles, so as to verify the correctness of the design of a system. Dynamics and reconfiguration in the communications is naturally supported by the features of the underlying $\pi$-calculus model. Being used, not only at the language level, but also as semantics, $\pi$-calculus appears suitable also for the application of techniques of automatic system verification.

Rapide, [LKA+95], is an event based model, where sets of events trigger and are produced by component reactions. Component behaviour is given in terms of condition-action rules, equivalent to finite state machines. Components are connected by connectors. The evolution of a system is represented in terms of the observation of the generated events. In particular, a relation of causality induces a partial order over sets of events (a true concurrent model), which can be constrained by a set of constraints. Connectors and constraints use an event pattern language for expressing conditions over occurring patterns of events. Correctness of a system is verified as the satisfaction, by the generated partial order, of a set of constraints.

Semantically, with its model of constrained true concurrency, Rapide proposal appears quite interesting. Semantic verification is based on the exploration of a finite state structure. Constraints over patterns of events are read as properties of the whole state space, and, like for other languages, verification is carried out by temporal model checking techniques, (see [Pnu81, Koz83], and Section 1.5.3). Finally, since Rapide is specifically designed for defining executable prototype of the architectures in an early stage of the development, it does not support the verification of evolving, or partial, system specifications.

Piccola [ALSN00] also distinguishes between components and connectors, but in this case connectors consists of scripts, which “code” the relations between the services offered by components. Interestingly, the (functional) scripting language does indeed rely on an underlying model based on a polymorphic version of $\pi$-
calculus. Authors, arguing that π-calculus appears as a too low level formalism to be successfully accepted by the application designer, have defined a more high-level language to make the practical use of their framework easier.

### 2.2.3 Future perspectives

It must be underlined that SOFTWARE ARCHITECTURE has been motivated within a context of closed systems. Nonetheless, the research in SOFTWARE ARCHITECTURE already covers significant issues for open systems, in particular the definition of suitable abstractions for components and their connections, e.g. interfaces and techniques for reconfiguration or modification of the communication topology, that is, issues related to autonomy and heterogeneity features. At any rate, they are oriented to define system development as an engineering process for assembling fully specified applications, which are totally contained within a controlled domain limiting the autonomy of components, practically not concerned with security or mobility issues, and which are verified against classical notions of correctness.

In this respect, the ability to deal with run-time behaviour reconfiguration, traditionally addressed by COORDINATION, appears to be a valuable direction for improving the architectural approaches, more oriented to static modeling. For instance, it could be interesting to export an event-based run-time reconfiguration mechanism, like the one discussed for the coordination language Manifold, to an architectural language like Rapide (described in the previous Section), which, even if based on an event model and provided with temporal logic based analysis, cannot naturally express dynamically evolving system specifications.

As also recognised by influential researchers in the field [Sha99] (whose position paper discusses, in more details, the following items), the open system paradigm imposes an upgrade of the goals and techniques which SOFTWARE ARCHITECTURE aims to achieve. In particular,

- extra information must be provided by the component abstractions, like the assumption on which they are based, and conditions under which they can be, possibly dynamically, integrated into a system. This is sometimes called metainformation or self-typing of a component,

- different notions of correctness, which are incremental, progressive and approximate, are needed. Verification of a system should be oriented at guaranteeing that “the system is sufficiently good for the tasks at hand,... and the analysis [is done] at a reasonable cost”,

- the incomplete accessibility of the system must be taken into consideration by suitable dynamic binding mechanisms as well as failure recovery strategies,

- security and mobility issues cannot be left out of consideration by system design.
These topics open new research directions for the development of software architecture in the age of network-centric computation. Some of these directions have been a source of motivation, and subsequently addressed in this thesis.

2.2.4 Discussion

Referring to the assumptions described in Chapter 1 about the model we intend to propose, we observe some coincident points with the scenario above mentioned. Behavioural concerns can be definitely interpreted as conditions under which components can be, possibly dynamically, successfully integrated. Consequently, behavioural interfaces are part of the advocated component metadata. Joining operations feature explicit dynamic bindings. Acceptability is an invariant and approximate property, aimed at guaranteeing that the system is “up to now” correct, and, under reasonable hypothesis, it can be effectively verified.

Finally, beyond the similarity with the possible developments of software architecture with respect to open systems, our proposal shares with existing ADL some similarities, as well as differences, that we briefly list in what follows.

According to languages like Leda and unlike Wright, and Rapide, the IP-calculus does not distinguish between connectors and components, and interconnections emerge by the coordinated interaction within a session. In this respect, our proposal does not aim at a clear correspondence with “lines and boxes” conceptual diagrams but rather at freely expressing the possible forms of interaction within a session of an open system. Moreover, we observe that a requirement for standard connectors may be limiting in an open system.

The relations of our work with Leda are strengthened also by the common aim of addressing behavioural issues in component composition. As claimed by Leda’s authors, we also believe that the choice of a reference model which provides abstractions for communication reconfiguration and mobility, as $\pi$-calculus or a derived process algebra, is mandatory when dealing with dynamic and open systems.

Both of us address the verification of behaviour conformance, but while they exploit concepts from the object-oriented paradigm, we are more oriented towards the incremental, progressive, approximate notion of correctness previously advocated.

The kind of verification as behaviour compatibility allowed by other ADLs, e.g. Wright, has also many contact points with our session verification. We have used finite model checking techniques, which are commonly adopted, in order to verify finite, temporally limited, behaviours of systems which are open, and may be partially specified.

Instead, the hierarchical composition of components and the corresponding modular definition of the interfaces, e.g. UniCon, suggest interesting directions for future development of our current proposal, e.g. how sessions can be nested? Which is the interface of a session (completions can be seen as an initial partial answer to this question)?
2.3 Component Based Software Engineering

2.3.1 Motivations

The main motivation for the development of COMPONENT BASED SOFTWARE ENGINEERING (CBSE) is reuse.

An outcome of the idea of modularity (one might say “component” in this context) is the principle “write once, run forever”. In an ideal world of components, once a component providing a functionality has been developed, it will be reused by everyone needing that functionality. By exploiting existing components, the development process can be improved in terms of time and cost. This, obviously, is subject to the development of a mature component market, so that CBSE development is subject to technological rules, as well as business rules. A mature component market is the one which also provides the means for individuating, downloading, certifying, and finally deploying the needed component into applications. In this sense, “software components are binary units of independent production, acquisition and deployment that interact to form a functioning system.” [Szy97].

The above definition highlights a couple of interesting points:

- components must be assumed to be binary units, both because they must be considered as black boxes for proper reuse, and because of business motivations: independent companies, obviously, do not intend to distribute the source code of the component they develop,

- the acquisition of a component implies a process, not addressed within this thesis, involving a potentially complex mechanism for describing, comparing and retrieving a component according to its desired properties,

- clearly, the deployment of a component into a functioning, i.e. correct, system made out of interacting components, presents similar problems, even if in a different context, to those that COORDINATION and SOFTWARE ARCHITECTURE attempt to solve, and that are covered by the open system metaphor (e.g. dynamic coordination of autonomous and heterogeneous computational entities, needs for the description and verification of the resulting system at a suitable architectural level, needs for secure certification of third-party components).

A substantial difference with the open system case is that, usually, the system specification is fully accessible and under control, (“architecture first”), and moreover, many argue that a fully mature component market requires that components are built according to an (industrial?) standard.

CBSE development originated from object-oriented modularity and then from its distributed version, based on the client/server interaction model. Components anyway go beyond objects, since objects appear too tightly coupled, their interaction is too dependent on the knowledge of the code of the other components and on its
variations (for example, with respect to polymorphism and inheritance, modifying a class/component may affect all the components which import an instance of that class or one of its subclasses. This sort of abstraction is called \textit{white box} interoperability, \cite{AFG02,BW98}). Moreover, objects do not provide sufficiently expressive interaction interfaces, late binding cannot naturally be achieved, and their behaviour is hard-wired into their code.

From the industrial side, several standards are competing on the market to become the reference model and setting the CBSE framework. Among the most important ones, we find the Microsoft COM architecture \cite{Cha96,Fei99}, the Sun JavaBeans \cite{SUNb}, and its extension Enterprise JavaBeans (EJB) which is a component framework supporting also component life-cycle, \textit{middleware} layers, like for instance CORBA \cite{Dol97,OMGa}, and more recently the framework .Net \cite{Mic}. All these models try to facilitate interoperability by providing abstractions which allow for composition. For example, COM equips its components with a standard mechanism by means of which a component exposes to its environment its interfaces, making methods accessible. A method-based interface is usually called a \textit{signature interface}. CORBA is an open distributed standard based on an \textit{Interface Description Language} (IDL), which allows for standardization, and on an Object Request Broker, to which components subscribe their IDL interfaces and which is responsible for transparently managing the invocation of one of its registered methods, working as a glue in composing different components spread over the network.

\subsection{Current trends}

The standard signature-based properties, also called \textit{functional properties}, that an IDL can describe, are not sufficient to fully characterise the interactive behaviour of a component. For instance, often IDLs only describe the offered methods without mentioning the ones on which they depend. According to \cite{LS97} (see also \cite{LS00}), other than functional properties, also \textit{behavioural}, \textit{synchronisation}, and \textit{quality of service} properties must be modeled. The behavioural ones are often expressed in terms of pre- and post-conditions on the availability of the services offered by a component, while the synchronization ones regard the concurrent aspects of interaction. Quality of services regards properties like performance, security, and availability of services. Analogously, \cite{CFTV99} individuates three levels of component interoperability: i) the signature level, ii) the protocol level, and iii) the semantical level, where the first two easily fit into the previous list, while semantical interoperability is recognised as the open and difficult problem of providing components with information about the semantics of the services they offer. Such information should enable automatic reasoning about the semantic of component composition, for instance during a coordination/negotiation phase, but currently this task is performed off-line by the system designer and it is out of the scope of our dissertation.

A major limitation of available component-oriented platforms is that they do not provide suitable means for describing and reasoning about the interactive behaviour
of concurrent components. The Concurrency Control CORBA service [OMGb], for instance, features a lock mechanism which is based on different access capabilities (read, write and update) over shared resources. The concurrent constructs of JavaBeans rely on the synchronized methods of Java (together with events and wait and notify primitives) for serializing concurrent threads during critical sessions [SUNb]. Even more enhanced approaches, like for instance the transaction support provided by EJB, force the component designer to explicitly manage the occurrence of a transition within the “functional” code of a component.

Another well-known approach is the so-called Design by contract [Mey92], where interaction runs on the basis of the verification of the compatibility of the respective pre- and post-conditions of the components involved. If compatible, such conditions become a contract to which components commit, as a warranty of the successful reciprocal interaction.

All these mechanisms do not seem to match the requirements that are widely considered as an index of quality for distributed software. In particular, they seem to force software designers to take into account too many low-level details, and they do not permit a clear separation of coordination from computation concerns, since the coordination “policies are generally hard-wired into application” code [CD99].

Several proposals have been put forward in order to enhance component interfaces with a description of their concurrent behaviour. Many of them are based on process algebras languages, such as π-calculus [MPW92], and extend interfaces with behavioural descriptions, such as behavioural types [NNS99] or role-based representation of behaviours [CFTV99].

These proposals allow one to prove correctness properties, such as absence of deadlocks, as well as to define compatibility relations, such as “the components can properly interact with one another” or “this component can be substituted with that one”. The techniques for reasoning on and verifying the resulting systems are typically co-inductive [JR97], based on the (stepwise) observation of the evolution of potentially non-terminating systems, and rely on the bisimulation relation or on modal (temporal) logics [Pnu81, CPS93, CW+96].

A limit of some of these approaches is the computational cost of proving such properties when they are interpreted as global properties of the whole system. In most cases the cost falls into the class of NP problems, and often it makes their practical usability difficult. In order to overcome this problem, some authors exploit local verification, like in [CPT99b], where the behaviour of components is partitioned into a finite set of roles, and global properties of the systems are inferred from local analysis of the relations among roles of corresponding components.

### 2.3.3 Future perspectives

As for the case of coordination and software architecture, CBSE is influenced by the advent of a network-centric, open model of computation.

As soon as the constraint of complete accessibility of the system is released, the
concept of reuse scales up from components to services. Instead of assembling a system by composing components acquired from the market, a new kind of architecture, in a more dynamic and volatile environment, consists in orchestrating services, which are available on the network, by dynamically linking them to an application which hence results spread over the network, open, not subject to a common domain, and only partially accessible.

Moreover, this kind of architectures would foster the expected shift from “business to client” (B2C) applications, where distributed applications interact with and are managed by human actors, to “business to business” (B2B) applications, where applications interact with other applications, dynamically selecting and exploiting the services that are offered.

A first proposal in this sense, is constituted by Web Services, [GGKS02], an architecture based on the concept of available service, where services are independently produced and published, together with a suitable description of their functioning (based on the XML format). Applications interact with services by locating them, possibly via a UDDI (Universal Description, Discovery, and Integration specification) repository, and by dynamically binding and using them, according to the paradigm publish, find, bind. The Web Service architecture is based on a standardised layered architecture, which is in charge of supporting the various phases of the process.

Not surprisingly, this proposed architecture, which is partially available but still under development in some of its essential aspects, calls for the definition of a new model of computation.

Clearly, the problems to be solved are the same as those which, in other forms, have been encountered in coordination, software architecture and CBSE, and are inherent to the open system metaphor. In particular, as suggested also in [AFG+02], it is necessary to integrate the benefits of the three approaches, in order to devise a model where coordination is clearly separated from computation, dynamic integration is properly supported by coordination interfaces and late binding techniques, autonomy, also in terms of unreliability, and heterogeneity of services can be dealt with, possibly by means of appropriate metadata abstractions, application design is engineered at an architectural level, and functional and non functional properties, like quality of services, certification, performances and security, can be properly described and verified by means of suitable verification techniques, which can be dynamically operated.
Chapter 3

Interaction Patterns and Sessions

Contents

3.1 Components and their environment .................................................. 36
3.2 Interaction Pattern calculus (IP-calculus) ........................................ 40
  3.2.1 Interaction Pattern syntax ...................................................... 40
  3.2.2 Interaction Pattern semantics .................................................. 44
3.3 Sessions ................................................................................. 47
  3.3.1 Accessing a session .............................................................. 49
  3.3.2 Session semantics ............................................................... 54
3.4 Correctness of sessions ................................................................. 58
  3.4.1 Correctness of closed sessions ............................................... 59
  3.4.2 Correctness of open sessions ............................................... 61
3.5 Verifying acceptability ................................................................. 65
  3.5.1 Completion standard form ...................................................... 66
  3.5.2 The algorithm $A$ (Acceptability checker) ................................ 72
  3.5.3 Constructing completions: examples ...................................... 79
  3.5.4 Complexity .......................................................... 82
  3.5.5 Correctness of the algorithm .............................................. 84

In this chapter we formalise the abstractions we use to describe and reason about open systems. A first introductory section discusses the assumptions we make and the hypothesis on which our approach is based. The following two sections introduce

1. the IP-calculus by means of which we describe the behaviour that a single component exhibits to its environment, and

2. the notion of session, a linguistic abstraction that describes the composition of interacting components into a system. In particular sessions comprise the mechanisms by means of which components can be connected together. Sessions uniformly model both closed and open systems.
The main aim of this dissertation is to provide an abstract methodology allowing for the verification of the properties satisfied by a session. In particular, we consider how to verify whether the composition of the single components yields a “correct” system, where correct means that all the components successfully accomplish their tasks. While this notion of correctness for closed systems reverts to usual dead-lock freedom, the case of open systems, which requires some assumptions on the “missing” components, is dealt with by introducing the notion of acceptability. Informally speaking, acceptability holds if all the components in the system can reach a successful termination point, possibly thanks to the contribution of components not (yet) in the system. In other words, acceptability is read as the absence of unrecoverable errors in the system as it appears “at the moment”. Acceptability, hence, is a property that must be preserved by, and which may usefully constrain, the access of new components in the system. The definition and effective verification of system correctness relies on the hypothesis, discussed in the next section, which guarantees the finiteness of the model to be analysed.

The algorithmic aspects of such verification are illustrated in Section 3.5, where an automated procedure to verify correctness and acceptability is shown and the correctness of the construction is proved. The algorithm implements model checking of the computational state space of a session. The search strategy introduced in this algorithm will be refined and applied to the algorithmic construction of adapting components, presented in Chapter 4, which are in charge of enforcing correctness by facilitating the interoperability of heterogeneous components.

Sometimes, throughout the technical presentation, the differences with other choices appeared in the literature are pointed out. The IP-calculus, together with the proposed methodology firstly appeared in [BBT01a] and were extended in [BBT01b].

### 3.1 Components and their environment

The term component refers to an autonomous computational entity that is designed to interoperate with other components in order to perform its tasks. Our research investigates the composition and coordination of components into a properly working system. In particular, we are interested in defining suitable abstractions that permit us to describe and reason about the behaviour of a complex system, starting from the behaviour of the single components that constitute the system.

It is hence necessary to describe the behaviour of the components “in isolation” and the mechanisms that allow their composition. The behaviour of each component is characterised in terms of what can be observed from the environment which hosts the component. More precisely, the behaviour of a component consists in the references that the component offers to the environment and the communications that the component performs through such references. Composition consists in appropriately connecting references, so that the behaviour that components exhibit through connected references properly matches.
This abstract definition of an open system as a partially specified system, offering “holes” for the access of other components, covers several different applicative domains, like, for example, systems that are partially specified since they are in an early (statical) design phase, or dynamically evolving applications based on mobile code. Our framework uniformly deals with all these cases of open systems, finding application in different scenarios.

Traditionally, components in their broader meaning are represented as black-boxes equipped with an interface that describes their features of interest for the “external world”. The advent of new computing scenarios, like distributed systems, component-based systems, web applications, and mobile code call for more expressive interfaces, providing information which permits to dynamically coordinate the autonomous activities of the components.

The IP-calculus, presented in this chapter, is designed to support behaviour concerns within component interfaces.

According to the approach explained in Chapter 1, the behaviour of a component is projected into a set of temporally finite fragments represented as interaction patterns. Each interaction pattern defines both the references (i.e. communication channels) offered to the environment and the behaviour, in terms of an expression of communication actions, that the component is ready to execute.

As explained in Chapter 1, this kind of “local in time” analysis is motivated by two principal reasons:

1. Dealing with the partial specification of open systems, for which the lack of complete information about the components that will eventually participate in the system makes the verification of global properties, i.e. properties regarding the whole life of a system, often unfeasible, and

2. reducing computational complexity, which reverts to the exploration of the finite structure (Proposition 3.1) of the computation of a session.

The basic abstraction for modeling the behaviour of a component in an environment is the access to a resource reference, where the term resource is intended in its broader meaning (from a communication channel, to the capability of accessing a secret, as intended in Chapter 5 in the context of security). Name-based formalisms naturally model resource access as the knowledge of a reference to the resource, viz. the name of the resource, and in particular, name-passing-based formalisms, like π-calculus, do it in a dynamic context.

In most of the cases, resources are encapsulated by components and a uniform way for accessing a resource is via a communication with the resource manager (and hence references are simply seen as channels). However, references can be read as, for example, object references in an object-oriented setting, or URLs (Uniform Resource Locator) in a web-based system, according to the system one may wish to describe.
Analogously, the semantical interpretation of the communication we adopt, see Section 3.3, can be refined to deal with the characteristics of a specific architecture. In fact, the semantics of communication, see Figure 3.5, requires that both the parties of a communication share the name of the channel and that they agree on the type of exchanged data: the agreement can depend on a given type discipline or on the specific features of the architecture under analysis (as, for instance, in Chapter 5, where agreement also means to share suitable cryptographic keys).

A component participates in a session according to one or more “proposed” interaction patterns and a given way of connecting the references of the patterns to those already available in the sessions.

Sessions are defined as a collection of interaction patterns sharing channels and interacting by communicating with each other according to the specified behaviour. Sessions are open, when they provide channels by means of which the interaction patterns that are not present at the moment into the session can possibly be connected to the session in the future. Otherwise they are closed.

A component can nevertheless join a closed session, possibly making it open again, but it can not share names (unless there are global names) with the already closed part of the session.

The finiteness of the amount of information represented by an interaction patterns facilitates the reasoning about such dynamic systems, like when verifying the compatibility of an interaction pattern with the session it is going to join.

It is important to point out that:

- Each pattern is independent of the others, which may belong to the same component, in particular they do not share a state. While this fragmented description of a component is not meant to describe global properties of a system, the pattern-based verification which we propose is well suited, as shown in the following, to prove the correctness of the composition of an interaction pattern of a component together with those that are present in a session. Intuitively, verifying interaction patterns corresponds to verifying the correctness of atomic transactions in the context of databases: in the same manner in which a system commits to a transaction only if all its requirements can be accomplished, an interaction pattern is allowed to join a session only if the behaviour it expresses can be proved correct with respect to the (currently “finite”) session.

Moreover, thanks to a subject reduction theorem (Theorem 3.1) a proved correct session will maintain its correctness through its possible evolutions, while access to potentially harmful interaction patterns will be denied.

- The IP-calculus alone does not describe the dynamics of the patterns as far as a component is concerned: there are not linguistic constructs which have
interaction patterns as objects, nor is it described how an interaction pattern is chosen among those that a component might exhibit. This also is motivated by generality, since different application architectures may require different constructs. Such architecture dependent constructs are however not essential for the issue of composing interaction patterns by coordinating the behaviours they represent.

For example, consider an architecture with migrating components which try to exploit the resources of different sites, like as it happens in an ad-hoc (wireless) network. In this case, components could be provided with reflective capabilities in order to negotiate their most suitable interaction patterns and the corresponding connections. The system would then accept only those patterns that have been automatically validated as compatible or at least harmless with respect to the session representing the local wireless site. Differently, in the case of an under-development system, or of a component-based application being updated by replacing some components, interaction patterns and connections are defined by the system designer. Our formalisation presents an unified view of the behavioural abstractions and connecting mechanisms for both the scenarios.

- The problem of determining a suitable connection among heterogeneous, possibly separately developed components, is a hard problem involving different disciplines, from software engineering to artificial intelligence. In general, we are not interested in studying how such connections can be determined, but rather in validating them, once they have been given, in relation to the interaction patterns and the session to which they refer. Moreover, as already said, depending on the specific architecture under analysis the process of definition of the connections may substantially vary, from engineer’s decisions to automatic negotiation.

A particular case is presented in Chapter 4, where it is discussed how, starting from the interaction patterns of two heterogeneous components and a high-level description of their intended connections, it is possible to devise a third component in charge of facilitating their interoperability, by “translating” to each other their communications.

Finally, it must be noticed that in an environment of autonomous components, the verification of the conformance of the actual behaviour of each component with respect to what is specified by its interface is a delicate issues. This issue concerns the security, and hence the correctness, of the whole system. It is currently an open problem under investigation, whose proposed solutions range from the use of wrappers forcing the untrusted components to behave according to their specification, to on-line verification of the code (proof carrying code, [Nec97]). This topic is anyway beyond the scope of this thesis.
3.2 Interaction Pattern calculus (IP-calculus)

In order to model interaction according to the hypothesis illustrated in the previous section, we have adapted \( \pi \)-calculus into a suitable process algebra, that we call IP-calculus. Interaction patterns are processes (or agents) of this calculus.

The choice of \( \pi \)-calculus as a reference model for developing the interaction pattern language is due to its recognised capability of naturally expressing many of the different aspects of open systems, like component linking, network reconfiguration, substitutability of components ensuring deadlocks avoidance, proper sharing of resources (in particular, of references, i.e. (channel) names). The usefulness of \( \pi \)-calculus has been illustrated for describing component models like COM [Fei99] and CORBA [GZ99], and architecture description languages like Darwin [MEK95] and LEDA [CPT99a].

According to \( \pi \)-calculus, patterns interact with each other by means of synchronous communication through shared channels. A channel is shared by two or more interaction patterns knowing the channel name. The behaviour of an interaction pattern is modeled by an expression combining communication actions via behavioural operators, like parallel or sequential composition.

### 3.2.1 Interaction Pattern syntax

The signature of IP-calculus consists of three sorts: data, communication actions and behavioural expressions.

Data are distinguished in basic data, variables and channel names. Basic data consist of the data proper of the applications. In a general formulation of the calculus we are not interested in distinguishing among application data, as well as in type checking them. When necessary, the framework will be extended in order to model peculiar aspects of a given architecture depending on application data. As mentioned, for example, when dealing with security aspects of composition, cryptographic data and primitives have been easily included in the calculus (Chapter 5).

Differently from \( \pi \)-calculus, where only names are used, variables (usually written uppercase) are used to explicitly indicate the missing data that a pattern acquires by interacting with the other patterns, either by receiving a datum via a communication, or by acquiring channel references from the environment. Names are shared...
via assignment to local (bound) variables: a mechanism that facilitates the separation of name spaces and hence information hiding among components. Channel names belong to a separate sort. Even if other data constructors could have been introduced, for the sake of simplicity, we consider here only the pairing constructor, so as to permit tuples of data (which makes the sort of data infinite). Other constructors will be introduced when needed for dealing with particular cases, like encryption primitives, a là spi-calculus [AG99], used in Chapter 5.

The syntax of data and communication actions is formally defined in Figure 3.1, where $X$ is a metavariable for variables, $B$ is a metavariable for basic data, $C$ is the sort of channels, i.e. either names ($Ch$) or (channel) variables ($X$), $(\_ , \_)$ is the pairing operator, $D$ is the sort of data.

Communication actions send or receive data through a channel: $in(C,D)$ is an input action receiving data $D$ through channel $C$, while $out(C,D)$ is an output action sending data $D$ through channel $C$. Metavariables $\alpha, \beta, \gamma \ldots$ range over communication actions.

**Example 3.1** The request made by an HTTP-client to receive a given page can be issued through a channel by means of the action

$$out(www.a.com, \text{page(index.html)}),$$

where $www.a.com$ is the name of a channel shared with the WEB-server and $\text{index.html}$ is the name of the required page (the method $\text{page(index.html)}$ belongs to application data). Correspondingly, a WEB-server may wait for a request by means of the action

$$in(www.a.com, \text{page(P)}),$$

where the name of the channel is again the address of the server and $P$ is a variable for storing the address of the required page. The two actions can synchronise, and hence communication happens, since they insist on the same channel and exchange compatible data. (Note how in this case, compatibility of data may include type-checking, i.e. the matching of the type of invoked and offered method).

A pair of compatible input/output action, as in the example, are referred to as dual actions, and the dual action of an action $\alpha$ is referred as $\bar{\alpha}$.

**Behavioural expressions** are built by composing communication actions and a distinguished silent action $\tau$ by means of the standard operators prefix ($_\cdot_\cdot$), parallel ($_||_\cdot$), and nondeterministic choice ($_+_\cdot$).

The silent action denotes an internal computational step that a component can perform independently of its environment, i.e. without synchronising with a communication action of any other component, and it is not observable from the environment.
The syntax of behavioural expressions is formally defined in Figure 3.2. Meta variables $P, Q, ...$ range over behavioural expressions. If $P$ is a behavioural expression, $\alpha.P$ is the expression that performs action $\alpha$ and then behaves as $P$, $P + Q$ can evolve either as $P$ or as $Q$, $P||Q$ can perform the actions of $P$ and $Q$ in any order, while $0$ denotes the empty behaviour.

It is worth noting here that, differently form $\pi$-calculus, dual actions of the same behavioural expression (process in $\pi$-calculus) do not synchronise with each other and communication does not happen, since we are not interested in modeling intrapatterns interaction, but rather interaction between different interaction patterns (hence, we have used the $||$ parallel operator, instead of the more standard $|$). In the full $\pi$-calculus, a silent action may indicate such an internal synchronisation step of a process.

In IP-calculus, where internal processes of a component do not synchronise, the presence of the silent action is motivated as a means to express local choices. The expression $\alpha + \beta$ can synchronise either with an action $\bar{\alpha}$ or an action $\bar{\beta}$ offered by another component in the environment. In this sense, this kind of choice, which depends on the interaction of more than one component, is called global. On the other hand, the expression $\tau.\alpha + \tau.\beta$, via an internal (autonomous) step can evolve either in $\alpha$ or $\beta$, hence autonomously deciding which action to synchronise on. This kind of choice is called a local choice. Local choices are necessary to model the autonomy of components.

Heterogeneity, instead, requires the separation of the name spaces of components. In practice, it is not reasonable to assume that heterogeneous components, possibly developed by different vendors, share a common set of names, e.g. know the method names of each other. It is hence assumed that, unless differently specified, when a component joins an environment, none of its channel names are known in the environment, nor does it know any of the channels already present in the environment. Then, names in every expression $E$ are local, unless differently specified (this can be recasted in $\pi$-calculus, by considering all the local names as explicitly restricted).

Anyway, since communicating requires the sharing of a channel name, components must be able to acquire channel references, in order to interoperate with each other. It is worth pointing out that interacting components can exchange names and consequently modify the communication network. Nonetheless, a suitable name-sharing mechanism is needed in order to initially connect together the channels of two or more components (as usual, connecting two channels is done by means of substitution, i.e. the names of different channels are mapped into the same shared name). Moreover, a clear definition of the mechanisms by means of which names
are shared facilitates the formal analysis of component-based systems.

Within a behavioural expression a channel may occur as

- a *local* channel, i.e. a channel name which is defined inside the scope of the expression,

- a channel that has to be *received* by a communication, i.e. a channel variable bound by an input action,

- a channel that has to be *connected* to a channel of another component in the environment, i.e. a channel variable in the behavioural expression not bound by an input action, which we call an *open variable*.

An interaction pattern consists of an explicit declaration of the channel variables it offers to the environment, the *open variables* of the patterns, and a behavioural expression specifying its interactive behaviour. We indicate the set of open variables with \((X)\) and a behavioural expression with \(E\).

Variables can be bound either by input actions or by open variables. The usual binding mechanism rules the scoping of variables with respect to input actions. The input action \(\text{in}(C,D)\) binds all the variables which occur in \(D\). The notion of input-free variables of an expression \((\text{ifv}(E))\), i.e. the set of variables not bound by input actions, is given by structural induction as (where \(\text{var}(D)\) is the set of variables occurring in \(D\)):

\[
\text{ifv}(E) = \begin{cases} 
\emptyset & \text{if } E = 0 \\
(\text{ifv}(F) \cup \text{var}(C)) \setminus \text{var}(D) & \text{if } E = \text{in}(C,D).F \\
\text{ifv}(F) \cup \text{var}(C) \cup \text{var}(D) & \text{if } E = \text{out}(C,D).F \\
\text{ifv}(F) & \text{if } E = \tau.F \\
\text{ifv}(F) \cup \text{ifv}(G) & \text{if } E = F + G \\
\text{ifv}(F) \cup \text{ifv}(G) & \text{if } E = F || G 
\end{cases}
\]

A variable \(V\) occurring in \((X)\), the open variables associated with a behavioural expression \(E\), binds the input-free occurrences of \(V\) in \(E\). A variable \(V\) is *free* in a behavioural expression \(E\) with respect to a set of open variables if there is an occurrence of \(V\) in \(E\) that is not bound (either by an input action or by open variables). An expression \(E\) is *closed* if it does not contain free variables.

**Definition 3.1 (Interaction patterns)** An interaction pattern is a process of IP-calculus of the form:

\[(X)[E],\]

where \(X\), the open variables of the pattern, is a set of variables and \(E\) is a closed behavioural expression (with respect to \(X\)).
Whenever necessary, the pattern can be annotated with both its name \( P \) and an identifier \( C \) of the component to which it belongs, like in \( P = (X)[E]_C \).

An appropriate binding mechanism for open variables is described by the semantics of the IP-calculus (Section 3.2.2). Informally speaking, components will be connected together in a session by unifying their open variables in a shared (common/global) name.

**Example 3.2** Let us consider a WEB-server. It repeatedly waits for a request over a known channel, namely its IP address, and then it either provides the requested page, or it replies with an error message. This pattern of interaction can be represented by the following interaction pattern:

\[
\text{(MyAdd)} \quad \text{(in(MyAdd, page(P))). (τ.out(MyAdd, a\_page.html).0 + τ.out(MyAdd, error).0)}
\]

The open variable \texttt{MyAdd} represents the channel offered by the server to its environment, to which a client component can be connected. After that \texttt{MyAdd} has been connected to a client channel, the server can receive a request for a page, by offering the “service” \texttt{page(P)} (the variable \( P \) represents the capability of the server to receive any value). Then, the server either sends back the requested page through the same channel, or it issues an error message. Note that in the chosen representation of the behaviour, the actual page and the kind of error sent are not relevant to interaction description and treated as constants. Silent actions, as well, “hide” the decision of sending either a page or an error message: it depends entirely on the server, which autonomously performs a local choice.

### 3.2.2 Interaction Pattern semantics

As usual for process calculi, semantics of IP-calculus is given as a Labeled Transition System (LTS), after the seminal idea of Structured Operational Semantics [Plo81], whose states represent processes and whose rules model how a state can evolve to another one. Transitions are labeled with information about the (observable) behaviour the processes exhibit. Rules may observe or abstract from the structure of the processes, and be combined or not with structural axioms, giving rise to a number of different approaches, yielding LTSs with different properties. A large number of papers have studied the properties of LTSs and their dependence on the structure of the rules and axioms defining the calculus, especially as far as the relation between the structure of the processes and properties like bisimulation is concerned, [dS85, Ber98, BIM95, GV92, BBB02]. For our purposes, labels are used to observe the action performed by an interaction pattern independently of its structure.

The semantics of the interaction of processes of IP-calculus is given by a pair of LTS. The first one models the intensional behaviour of the interaction patterns in
3.2. INTERACTION PATTERN CALCULUS (IP-CALCULUS)

\[
\begin{align*}
(E | F) & \equiv E | (F | G) \\
E | F & \equiv F | E \\
E | 0 & \equiv E \\
(E + F) + G & \equiv E + (F + G) \\
E + F & \equiv F + E \\
E + 0 & \equiv E
\end{align*}
\]

Figure 3.3: Structural equivalence axioms.

isolation, independently of the environment in which they will operate. As said, differently from π-calculus, parallel components of the same interaction pattern do not communicate with each other, and hence all the actions an interaction pattern may execute are observable, as labels, from its environment. The second LTS, expressing the interaction between the patterns in the same session, is built upon (the labels of) the first one, and models how patterns communicate by synchronising on dual actions. Moreover, a rule of session semantics models how interaction patterns can join a session. The rule is at the present partially specified since it will be used to check access conditions for patterns, as it will be explained in the following section.

Semantics of stand alone interaction patterns is given by the LTS \( \rightarrow \), defined in Figure 3.4. The rules correspond to the classical rules for π-calculus, the main difference being the absence of communications at this level of abstraction. The relation \( \rightarrow \) is defined up to structural congruence \( \equiv \) which requires that \( + \) and \( | | \) are associative monoidal operators with 0 as neutral element, as defined in Figure 3.3, and that expressions are equivalent up to \( \alpha \)-renaming, [MPW92]. An expression consisting of a prefix can execute the prefix action (rules (\( \tau \)) and (\( \text{act} \))), the parallel composition of two expressions can evolve according to both the expressions (rule \( \text{par} \) together with \( \text{struct} \)), while the sum of two expressions can non-deterministically evolve according to one of the two expressions (rule \( \text{sum} \) together with \( \text{struct} \)). The rule \( \text{struct} \) allows for a more compact presentation of the semantics, by making rules that correspond to structurally equivalent expressions not necessary.\(^1\) Last rule \( \text{pat} \) lifts actions as label from behavioural expressions to interaction patterns. As expressed by the rules, an interaction pattern alone does not modify its open variables.

As usual, \( \rightarrow^+ \) and \( \rightarrow^* \) indicate respectively the transitive and the reflexive and transitive closure of \( \rightarrow \).

As it emerges from the presented syntax, the main differences of IP-calculus with respect to the full π-calculus are:

1. Absence of recursion, according to the premises explained in Chapter 1, we model the interaction in an open system by observing transactions, where components coordinate their finite patterns of interactions. In this settings, patterns are not allowed to exhibit infinite, recursive behaviours.

\(^1\)For example, the transition \( E + F \xrightarrow{\alpha} F' \) can be obtained by rules \( \text{sum} \), \( \text{struct} \) and by the equivalence \( E + F \equiv F + E \).
2. **Explicit (syntactical) treatment of open composition**, in terms of the declaration of the channel references as open variables. This way of proceeding resembles abstraction of polyadic π-calculus [Mil93]. In that calculus, an explicit abstraction is given to uniformly bind both local (restricted) and input-bound names. This abstraction mechanism is then used to both connect and let different processes communicate. Connections, as in our case (see join operation of Definition 3.4), happen via channel names, rather than process names composition. Differently from the more general polyadic π-calculus, in our framework communications and connections are not represented by an uniform name abstraction, and name abstraction can not be structurally nested into processes, since sub-processes do not interact, neither by communicating or by connecting with each other. This also simplifies the representation of a system. Finally, it is not required that all the open variables are instantiated together, as happens to those of an abstraction, so that a, possibly partially connected, interaction pattern may wait for more partners to come, and then bind some other of its open variables to them.

3. **Implicit restriction of local names**. All the names of an interaction pattern are local, according to encapsulation principles. This can be achieved by standard α renaming, as required by the operation of composing components into systems, see Definition 3.4. Note that in π-calculus, restriction, together with recursion (or replication in the form of ! operator), allows for the dynamic creation of infinitely many new names. Interaction patterns, instead, can only use the finite set of names they initially declare plus the finite set of the names of the channels which are established when they are connected to each other.

**Remark 3.1** *The possibility of creating new names is an essential feature for any*
Turing complete formalism, from which it follows that IP-calculus is not Turing complete. For example, a pattern is not able to count further than up to a fixed number. This “incompleteness” is anyway coherent with the hypothesis adopted for our model of interactive systems.

Moreover, we recall here that there is not a clear agreement about the relation among computation and interaction. Many authors from different fields believe that a clear separation of the two concerns is a valuable approach in understanding interactive systems, distinguishing computational (reflective) steps of a component from those in which components interact with their environment (in the field of Coordination Languages see, among the others, [ACG86, PA98], while, for instance in the field of Artificial Intelligence, this is a settled principle, where intelligent agents are often seen in terms of a perceive-think-act cycle). Other authors strengthen further the principle by calling for the definition of a new computational model, since they suppose that Turing Machine is not suitable to properly model interaction concerns, [Weg98].

### 3.3 Sessions

Open systems are usually described in process calculi by the notion of context or coordinator, i.e. a term of the calculus having variables (or holes) ranging over (closed) processes, like $C[X]$ or, in the hole-style, $C[\_]$. For instance, the context $n[X] | m[Y]$ represents, in an Ambient like calculus [CG00], the parallel composition of two open ambients, namely $n[\_]$ and $m[\_]$, which run in parallel and contain not yet specified processes.

The problem of reasoning about contexts as descriptions of open systems still lacks a completely satisfactory answer, and in many cases lifting the process-level techniques is not enough. For example, defining context equivalence in terms of processes equivalence, by considering the contexts $C[X]$ and $D[X]$ equivalent if, for any process $p$, the processes $C[p]$ and $D[p]$ are equivalent, brings to the usual difficulties due to universal quantification and the infinite branching it may cause. A way to address this problem, as discussed in [BBB02], is to revert to symbolic analysis, partitioning transitions into classes (symbolic transitions), hopefully finite in number. Each class represents all the transitions involving those processes which, because of their (structural or behavioural) features, can be collected together. Typically, the behaviour exhibited by the processes of the same class can be represented by (minimal) conditions over the actions of the processes and of the context, see [Sew98, LM00, BBB02]. Other authors define context semantics by enriching transitions with annotations obtained by lifting the algebraic structure of the terms, e.g. [LX90].

According to the framework we propose, we do not aim at dealing with general contexts, but rather with contexts that have a quite simple, flat, structure built only by means of the parallel composition of interaction patterns. Moreover, openness
of our contexts is not given by variables which range over processes, but rather by variables which range over (channel) names, i.e., the open variables of an interaction pattern. The approach “process as variables”, instead, does not make explicit the connections of components, but rather requires the presence of global names. For instance, a process instantiating an ambient-like context as $n[X]$ will be able to interact with the environment, e.g., by means of an exiting action $\text{out } n$, depending on its previous knowledge of the name $n$. Note how this requirement cannot be naturally described by the instantiation of the variable $X$ with the process. Differently, in our formulation, composition does not consist in the instantiation of a hole in a structured context, but in the explicit sharing of names via the instantiation of open variables.

In our framework, what has been up to now generically called environment, is called session, so as to explicitly mark the differences with the more standard notion of context or coordinator and to remark its finitary temporal dimension, recalling the finite transactions on which we concentrate in studying the properties of a system. The formalisation of sessions that we present here has been originally introduced in [BBT01a] (with the name of context).

Sessions apply to both closed and open systems, permitting us to express different correctness properties inside an uniform setting. In this chapter correctness is understood as the possibility for the components in the session to successfully accomplish their tasks. In the case of closed sessions, correctness reduces to standard deadlock freedom.

The notion of session represents an abstraction of the boundaries of a component-based computation, and it may model different architectures, like security domains, dynamically reconfigurable applications, distributed applications or sites accepting migrating code.

Simply, a session is a set of interaction patterns.

**Definition 3.2 (Session)** A session $S = \{(X_1)[E_1], \ldots, (X_n)[E_n]\}$ is a possibly empty set of interaction patterns.

Sessions are determined by the interaction patterns they contain and by the way in which their interaction patterns have been connected together. The union of the open variables of the patterns in a session are connections offered to new interaction patterns joining the session. The formal definitions of open and closed session follow.

**Definition 3.3 (Open and closed session)** A session $S = \{(X_1)[E_1], \ldots, (X_n)[E_n]\}$ is closed if $\bigcup_{i=1}^{n} X_i = \emptyset$. It is open, otherwise.

A session can evolve either because of the joining of a new interaction pattern or because of the communications among interaction patterns already in the session.
3.3.1 Accessing a session

An interaction pattern accesses a session by establishing connections with the other interaction patterns in the session. Connections are created by mapping open variables of two or more patterns into the same new channel name, according to a given mapping. Acceptance of the incoming interaction pattern may be subject to the satisfiability of correctness properties by the session which would result from the insertion of the new pattern. In particular, we have addressed the case in which acceptance prevents incoming patterns from introducing unrecoverable deadlocks, as it will be made more precise by the semantics of sessions.

Incoming interaction patterns must be variable-disjoint with the patterns already present in the session (all the variables of a closed interaction patterns are bound and can be, if necessary, renamed (α-conversion)). Moreover, according to the hypothesis of separation of the name spaces of components, interaction patterns are generally assumed to be also name disjoint. When, differently, interaction patterns share common names, it will be explicitly stated. For example, this will be the case when addressing security issues in Chapter 5, where untrusted communications have been modeled by a channel accessible by any component in the environment. When interaction proceeds inside a session, further channel names can be shared by means of communication, as typically happens in π-calculus.

An interaction pattern may also join a closed session (by the empty mapping $\epsilon$), but, under the assumption of name-disjointness, it will not influence the interaction of the closed group of components already in the session.

Open sessions can then be seen as partial specified sessions, whose future history depends also on the components that will eventually join it and that are at the moment unknown. Note that this partial specification can be due to the dynamism of the system, but also to its partial construction in a statical phase of modular development and testing.

Example 3.3 (Mapping) Mapping definition is in the general case a difficult, domain dependent problem that is beyond the scope of this dissertation. A simple example may anyway help in making more precise the nature of the problem.

Mapping mechanism can be illustrated by referring to Example 3.2, where a WEB-server offers a channel, namely the open variable MyAdd, by means of which a component may require a service. Consider a WEB-browser that want to access a session containing the WEB-server in order to download some pages. The WEB-browser may have an open variable, WebPort say, in its interaction pattern:

$$(\text{WebPort}) \ [\text{out(} \text{WebPort, page(index.html)}\text{)}. \text{in(} \text{WebPort, Data}\text{)}.0]$$

In practice, the matching is easily done since both components adhere to the same protocol, so that both of them agree on the meaning of an IP address. According to the practice, the matching of the server’s open variable with the corresponding open variable of the client is based on the knowledge by the client of the IP address of the server. In our abstraction, this will be modeled by matching the two different open
variables of the components into the same fresh channel name, $ipc$ say, obtaining the closed session:

$$\{ (\text{in}(ipc, \text{page}(P))). (\tau.\text{out}(ipc, \text{page}) + \tau.\text{out}(ipc, \text{error})],$$

$$\{ \text{out}(ipc, \text{page(index.html)}). \text{in}(ipc, \text{Data}).0 \} \},$$

Another case of dynamical mapping of names, can be observed in the practice in the middleware layer of a distributed architecture, like for instance in the case of the Object Request Broker (ORB) of the CORBA architecture [OMGa]. The ORB provides the appropriate reference of a service provider component. The connection between a client and the server provider can be modeled in two steps: the connection of the client with the ORB, and then a direct communication between the client and the server provider. In this case, differently from the previous example where the WEB-client is supposed to know the WEB-server address, references are shared dynamically. Nonetheless, the basic mechanism used, i.e. sharing of common references, is the same. Example 3.5 will provide an interaction pattern for an ORB.

The operation of component access to a session has not been introduced directly into the language of the interaction patterns, but it appears as a transition rule of session semantics, the $join$ operation. Keeping mapping mechanisms apart from the calculus, permits us to abstract from the problem of determining how components need to be connected, when describing interaction and composition, so as to focus on whether a given composition is admissible with respect to certain (correctness) properties, without dealing with how the mapping has been determined.

The calculus may anyway be easily extended with a primitive $join$ action, when required by the system analysed. For example, describing mobile code, which migrate throughout sites which provide the support of such migration, one may assume that the support is able to determine the needed matchings, and encapsulate such mechanisms into the interaction patterns language.

Starting from the empty session, the $join$ operation permits sessions to be (dynamically) constructed. Given a session and a mapping connecting open variables to channels, the $join$ operation rules how a new interaction pattern, name-disjoint with those in the session, can access the session.

**Definition 3.4 (Join)** Let $S = \{(X_1)[E_1],..., (X_n)[E_n]\}$ be a session, and let $(Y)[F]$ be an interaction pattern (variable- and name-disjoint with the session). Let

$$\gamma : \bigcup_{i=1}^n X_i \cup Y \rightarrow Ch$$

be a partial mapping. Then:

$$\text{join}((Y)[F], \gamma, S) = \bigcup_{i=1}^n \{(X_i \setminus \text{dom}(\gamma))[E_i\gamma]\} \cup \{(Y \setminus \text{dom}(\gamma))[F\gamma]\}.$$
Applying a mapping to a behavioural expression corresponds to applying all the (ground) substitutions of the mapping to the expression. Substitution is defined in the expected way, with renaming possibly used to avoid capture of free names. With \([a/x]\) we indicate the substitution of \(a\) in the (free) occurrences of \(x\). Hence, substitutions of the mapping \(\gamma\) are applied to the occurrences of the open variables which are free in the behavioural expressions of the interaction patterns in the session. After having been connected, open variables are not open anymore, i.e. they can not be connected again to different channels.

Note that the join operation can connect open variables of different interaction patterns into the same channel name, realising a channel shared by more patterns. A bit of care is necessary in choosing the mappings, since assigning a variable is a “destructive” operation regarding its openness, and it is hence necessary to assign together at the same step all the open variables that must be mapped into the same channel (this is simply obtained by delaying the assignment until all the patterns containing the selected open variables are in the session).

Analogously, the simultaneous joining of more patterns can be reduced to the sequential joining of a pattern at a time. Informally speaking, a join operation of the form

\[
\text{join}(P_1, \ldots, P_n, \Gamma, \mathcal{S}),
\]

where \(P_i\) are interaction patterns and \(\Gamma\) is a mapping regarding their open variables and the open variables of the session \(\mathcal{S}\), can be equivalently replaced by a sequence of operations

\[
\text{join}(P_1, \gamma_1, \text{join}(\ldots, \text{join}(P_n, \gamma_n, \mathcal{S}))\ldots).
\]

yielding the same session (provided that each \(\gamma_i\) correctly assigns variables that have to be mapped into the same channel, as explained). On the other hand, the (repeated) use of the simultaneous join can be trivially used to simulate the step-wise formulation, again obtaining the same session. Properties that can be verified on a simultaneously constructed session, can also be eventually verified on the step-wise constructed session, which, in the end, contains the same amount of information. Hence, the simpler step-wise formulation of the operator has been adopted.

The two following examples illustrate the construction of a closed and an open session.

**Example 3.4 (Closed session)** Let us consider a generic client-server interaction inside a session. The client interrogates the server and then either it waits for an answer, or, after a time-out say, it autonomously decides to send an aborting message. Such a client can propose to the session the following interaction pattern:

\[
\text{CLIENT = (S) [ out(S,query). (in(S,Answer).0 + \tau . \text{out}(S,\text{break}).0) ]}
\]

where \(S\), the only open variable of the pattern, is intended to be a connection to a server.
A server is ready to offer a service. It waits to receive a request and then it either sends an answer or it is able to react to the aborting message \texttt{break} by closing the protocol. The server offers only one channel to its environment, on which it receives the request and then forwards the answer unless an aborting message is received. Its interaction pattern is:

\[
\text{SERVER} = (C) \left[ \text{in}(C, \text{Query}). \ (\text{out}(C, \text{answer}).0 + \text{in}(C, \text{break}).0) \right]
\]

Although simple, the example illustrates the use of \(\tau\) and + to model internal (local) choices, in fact note that differently from the choice of the client, the one of the server is determined by the environment with which the server interacts. The example also highlights the very idea of interaction patterns as a means to express recurrent fragments of interactions: each time a server interacts with a client, it will do it following the same scheme, a sort of behavioural fingerprint.

The client and the server can be composed in a closed session by connecting their open variables to the same channel name \(n\):

\[
S = \text{join}(\text{CLIENT}, [n/S, n/C], \text{join}(\text{SERVER}, [], \{\})) =
\begin{align*}
\{ \ &() \left[ \text{out}(n, \text{query}). \ (\text{in}(n, \text{Answer}).0 + \tau.\text{out}(n, \text{break}).0) \right], \\
\ &() \left[ \text{in}(n, \text{Query}). \ (\text{out}(n, \text{answer}).0 + \text{in}(n, \text{break}).0) \right] \}. \\
\end{align*}
\]

Informally speaking, this simple session appears “properly constructed”, because of the “compatibility” of the two patterns. On the other hand, a closed session where a different interaction pattern of a server, not capable of handling aborting messages, has been connected to \(\text{CLIENT}\):

\[
\begin{align*}
\{ \ &() \left[ \text{out}(n, \text{query}). \ (\text{in}(n, \text{Answer}).0 + \tau.\text{out}(n, \text{break}).0) \right], \\
\ &() \left[ \text{in}(n, \text{Query}). \ \text{out}(n, \text{answer}).0 \right] \},
\end{align*}
\]

appears as a session that may run into trouble in some points of its evolution.

**Example 3.5 (Open session)** In the CORBA architecture, [OMGa], a component can require a service (e.g. a reference to an object) by means of the so-called ORBs (Object Request Broker). Let us consider the interaction pattern of an ORB. Its task is to locate services that other components in the session offer. When it receives a request for a service, it returns an appropriate channel by which the service can be referenced. We model an ORB that can choose between three references it has in its repository, by means of the following interaction pattern \(\text{ORB}\):

\[
\text{ORB} = (C) \left[ \text{in}(C, Q). \right. \\
\left. (\tau.\text{out}(C, ch1).0 + \tau.\text{out}(C, ch2).0 + \tau.\text{out}(C, ch3).0) \right].
\]

After receiving the request, which is stored by the variable \(Q\), \(\text{ORB}\) selects an appropriate reference, via an internal computation and sends it back. If the \(\text{ORB}\) component wants to serve another querying component or more components concurrently, it
joins to the session more instances of the pattern. Unless assuming global names, ORB must have previously acquired the channel references ch1, ch2 and ch3 either by means of suitable open variable matchings or by communication among service providers and ORB.

An interaction pattern of a component that intends to require a service, and then to reference it, is:

\[
\text{REQUEST} = (S) \ [\text{out}(S, \text{request}).\text{in}(S, \text{CH}).\text{out}(\text{CH}, \text{call\_service}).0].
\]

REQUEST does not know a reference to the service until it is communicated by an ORB. Note how, in the spirit of \(\pi\)-calculus, the topology of the network may be reconfigured by interaction.

Let us extend now the example so as to explicitly model how service providers register themselves to an ORB component. A first component registers itself by providing an open variable, i.e. Q1, to be used as the channel on which the service request will be received. The second one has an open variable, i.e. S2, used to communicate to ORB the channel ch2 on which the service request will be received:

\[
\begin{align*}
C1 &= (Q1) \ [\text{in}(Q1, \text{REQ1}).0], \\
C2 &= (S2) \ [\text{out}(S2, \text{ch2}).\text{in}(\text{ch2}, \text{REQ2}).0].
\end{align*}
\]

The new version of ORB offers to its environment two open variables, R1 and R2, for receiving service registration and one, C, for receiving a request and forwarding a reference. R1 is used as a channel to directly referencing the service, while R2 is used to receive the service referencing channel. After a request, ORB decides to which service to redirect it. In the case in which it chooses the first one, it has only to forward the reference obtained when the first provider joined the session. Otherwise, it first receives the reference (through channel R2) and then it can forward it.

\[
\text{ORB} = (C, R1, R2)[(\text{in}(C, \text{Request}).\tau.\text{out}(C, R1).0 + \tau.\text{in}(R2, \text{CH2}).\text{out}(C, \text{CH2}).0)].
\]

Connecting ORB with the pattern C1 by means of the mapping \([\text{ch1}/R1, \text{ch1}/Q1]\), we obtain the following open session:

\[
\{ () \ [\text{in}(\text{ch1}, \text{REQ1}).0], \\
(C, \text{R2}) \ [(\text{in}(C, \text{Request}).\tau.\text{out}(C, \text{ch1}).0 + \tau.\text{in}(R2, \text{CH2}).\text{out}(C, \text{CH2}).0)] \}
\]

Note that, if on the one hand there is not any constraint about new patterns joining the session, on the other hand, if a pattern intends to connect to the channel (variable) C, its behaviour is expected to be compatible with that of its partner ORB over that channel. If the pattern REQUEST joins the session by means of the mapping \([\text{ch0}/S, \text{ch0}/C]\), we obtain:
an open session in which one would expect that, thanks to a proper mapping, interaction involving \texttt{ORB}, \texttt{REQUEST} and \texttt{C1} would correctly happen. In the following, we will show how to formally verify and enforce this sort of “potential correctness” along the evolution of a session.

### 3.3.2 Session semantics

Semantics of sessions is given by the transition system $\rightarrow$ of Figure 3.5 that models interaction among interaction patterns and their access to a session.

As far as interaction is concerned, the transitions system is based on lifting the observable behaviour of interaction patterns from the interaction pattern semantics into session semantics, and appropriately combining them.

This two level semantics, permits the session semantics to be modularly adapted to different calculi for interaction patterns, which may be needed in analysing specific systems (provided that the notion of observable behaviour allows a compatible and analogous notion of matching dual action). This reflects, at a semantic level, the idea of separating the coordination (communication) layer from the one regarding autonomous activities of components, as pursued, for instance, by the coordination language Manifold [BaD00].

Interaction inside sessions is modeled by synchronous communication actions occurring over a channel: input (structured) data and output data must match. Trivially, the communication channel must be specified, i.e. communication does not happen through channel variables.
Rule *comm* models synchronous communication: two patterns, $P_i$ and $P_j$, ready to synchronise on a pair of dual actions, respectively, can communicate if input data (e.g. a formal method invocation) match output data (e.g. an actual method call, with the “right” parameters).

Matching of an input datum $d'$ with an output $d$ is defined as the existence of a *ground substitution* $\sigma$, such that $d'\sigma = d$, where $=$ is syntactic equality. Note that the sent message, coherently with the hypothesis of closeness for interaction patterns, are ground. As said, the notion of matching can suitably be extended to encompass type checking or other features relevant for the system under analysis. As an example, in Chapter 5, where a cryptographic system has been embedded in the calculus, the notion of matching implements encryption and decryption.

The term $P\sigma$ denotes the application of a substitution $\sigma$ to an interaction pattern $P$, and it is defined as expected: $(X)[E]\sigma?>>>(X)[E\sigma]$. In this way data received are stored in the receiving pattern.

Communications in distributed systems over wide-area networks, like the majority of the cases of mobile code or web-based applications, are typically asynchronous. In fact, network delays and possible failures make communication unreliable, let alone synchronous, [Car99]. Our choice of using a synchronous calculus relies on the desire to abstract from the possible delays and errors due to the network, so as to concentrate on the compatibility of the interaction patterns.

Rule *silent* models internal, non interacting, computational steps of a single interaction pattern. As shown in examples, silent actions play a crucial role in modeling local choices, i.e. states that a pattern can reach independently of the synchronisations with the other patterns.

The rule *join* models the evolution of a session due to the joining of a new interaction pattern. Given the session $S$, a pattern $P$ and a mapping $\gamma$, the session $S'$ is the new session obtained according to the *join* operation. The transition is subject to the satisfaction of a, at the moment, not further specified property $P(S, P, \gamma, S')$, involving all the arguments (and the result) of the join operation. As it will be shown in Section 3.4.2, an appropriate definition of the property is intended to deny the access to the interaction patterns that may spoil the correctness of the session (or other possible properties of interest). Notice that a “bad” $\gamma$ might yield a “wrong” session where the desired property does not hold. Some remarks are worth to be made here:

1. The rule models an operation, i.e. *join*, that does not belong to the language. One could have introduced a primitive action $\text{in}(S, \gamma)$ together with the explicit notion of sessions in the calculus, in the style of calculi with ambients, see for example [CG00], obtaining an explicit description of mobility. By abstracting from a specific mobility model, we aim at modeling the more general concept of composition, concentrating on the problem of how a component can join a context (session), given the connections among the component and the context, no matter how they have been determined. Specialising our generic
model into a specific one, having for example locations and access restrictions for mobile code, appears quite straightforward by specialising the join operation and extending the calculus as necessary.

2. The join operator permits a session to evolve through infinitely many steps by the contribution of infinitely many new interaction patterns that join the session. This does not invalidate our hypothesis of studying finite structures since:

- we will reason about properties regarding the structure of a session at a given instant without considering its possible evolutions due to the joining of other interaction patterns,
- we will show how to preserve correctness properties along the (infinite) life of a session by finitely verifying the compatibility of the joining interaction patterns with those already in the session.

3. The property $P(S, P, \gamma, S')$ of rule $\textit{join}$ is meant to express behavioural properties of the interaction patterns in a session and to enforce correctness by restricting the access to a session to the interaction patterns that exhibit a potentially dangerous behaviour. It may predicate over the session, the joining pattern and their connections. More in general, the property may also enforce the satisfaction of other desired behavioural properties, like access resource control or security properties (see Chapter 5).

While ad-hoc logics can be defined according to the kind of desired properties, the way in which verification is performed is generally based, as in our case, on finite model checking, exploiting the benefits of analysing finite structures only (see [CW+96]).

An initial attempt to enforce properties by means of only checking the kind of connections made, i.e. the mapping $\gamma$, has been studied in [BBFT02] with the aim of reducing the model checking of properties over computations to the more efficient verification of constraints over mappings.

We indicate with $\rightarrow^+$ the transitive closure, and with $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$, while $\not\rightarrow$ stands for the absence of a rule to apply, so that $R : S \rightarrow^* R$ is a session reachable from $S$, while $R \not\rightarrow$ is a deadlocked session. Labels of the $\rightarrow$ relation may be omitted, when it is not necessary to distinguish the actions which permit the transitions to be fired. A sequence of sessions $\Delta = S_0 \ldots S_i \ldots$ such that $\forall i. S_i \rightarrow S_{i+1}$ is called a trace of the session $S_0$. Sometimes, to make clear that we are referring to traces in which $\textit{(join)}$ rule is not applied, i.e. evolutions of a session without the contribution of new joining interaction patterns, we will use the symbol $\rightarrow_\gamma$.

**Example 3.6** The open session of Example 3.5, unless a new pattern joins it, is forced in its first step:
\[
\{ () \uparrow \text{in(ch1, REQ1),0 ],}
\]
\[
(R2) \uparrow [(\text{in(ch0, Request).} \\
\quad (\tau \cdot \text{out(ch0, ch1).0 + \tau \cdot \text{in(R2, CH2).out(ch0, CH2).0)})),]
\]
\[
() \uparrow [\text{out(ch0, request). in(ch0, CH). out(CH, call_service).0 } ]
\}
\]
\[
in(ch0,Request),out(ch0,request)
\]
\[
\{ () \uparrow \text{in(ch1, REQ1),0 ],}
\]
\[
(R2) \uparrow [\tau \cdot \text{out(ch0, ch1).0 + \tau \cdot \text{in(R2, CH2).out(ch0, CH2).0)}],
\]
\[
() \uparrow [\text{in(ch0, CH). out(CH, call_service).0 } ]
\}
\]
\[
Now everything depends on ORB: it can not communicate along R2, unless a new pattern connects to that channel by joining the session, but it can process the \text{request} received and autonomously decide for one of the two alternative it has. If it decides for the first possibility:
\]
\[
\tau \rightarrow
\]
\[
\{ () \uparrow \text{in(ch1, REQ1),0 ],}
\]
\[
(R2) \uparrow [\text{out(ch0, ch1).0 }],
\]
\[
() \uparrow [\text{in(ch0, CH). out(CH, call_service).0 } ]
\}
\]
\[
the session has only one trace towards the successful state
\]
\[
\{ () \uparrow [0],
\]
\[
(R2) \uparrow [0],
\]
\[
() \uparrow [0 } ]
\}
\]
\[
Otherwise, if the second alternative is chosen:
\]
\[
\tau \rightarrow
\]
\[
\{ () \uparrow \text{in(ch1, REQ1),0 ],}
\]
\[
(R2) \uparrow [\text{in(R2, CH2).out(ch0, CH2).0}],
\]
\[
() \uparrow [\text{in(ch0, CH). out(CH, call_service).0 } ]
\}
\]
\[
then interaction is clearly stuck. Let us suppose that the interaction pattern C2 of Example 3.5 now joins the session with the mapping [ch3/R2, ch3/S2], making it closed:
\]
\[
j(C3,[ch3/R2, ch3/S2])
\]
\[
\{ () \uparrow \text{in(ch1, REQ1),0 ],}
\]
\[
() \uparrow [\text{in(ch3, CH2).out(ch0, CH2).0 }],
\]
\[
() \uparrow [\text{in(ch0, CH). out(CH, call_service).0 } ],
\]
\[
() \uparrow [\text{out(ch3, ch2).in(ch2, REQ2).0 } ]
\}
\]
so that the following steps are possible:

\[
\begin{align*}
&\text{in}(ch3, CH2), \text{out}(ch3, ch2) \\
&\implies \\
&\{ (\) [ in(ch1, REQ1).0 ], \\
&\) [ out(ch0, ch2).0 ], \\
&\) [ in(ch0, CH). out(CH, call\_service).0 ], \\
&\) [ in(ch2, REQ2).0 ] \} \\
&\implies \\
&\text{in}(ch0, CH), \text{out}(ch0, ch2) \\
&\implies \\
&\{ (\) [ in(ch1, REQ1).0 ], \\
&\) [ 0 ], \\
&\) [ out(ch2, call\_service).0 ], \\
&\) [ in(ch2, REQ2).0 ] \} \\
&\implies \\
&\text{in}(ch2, REQ2), \text{out}(ch2, call\_service) \\
&\implies \\
&\{ (\) [ in(ch1, REQ1).0 ], \\
&\) [ 0 ], \\
&\) [ 0 ], \\
&\) [ 0 ] \}.
\end{align*}
\]

Clearly, the pattern (\) [ in(ch1, REQ1).0 ] can not successfully terminate unless another pattern in the session is able to communicate through the channel ch1, that is not the case, nor will it be in the future if the name ch1 is local to the pattern and (hence it can not be shared by any joining pattern).

In the following, we will define a notion of correctness for closed and open sessions, suitable to prevent a misbehaviour like the one of the session in the previous example.

### 3.4 Correctness of sessions

As explained in Section 3.1, one of the main interests of our proposal is the possibility of practically verifying properties of interactive systems. Indeed the fact that interaction patterns (and hence sessions) express only finite fragments of interaction bounds the state explosion problem typical of this kind of analysis, and situates our analysis in the finite model checking field. This notion of finiteness is formally expressed by the following Proposition that applies to the traces of a session that is not joined by any new interaction pattern.

**Proposition 3.1** Let \( S \) be a session. The set \( \{ S' \mid S \xrightarrow{\pi} S' \} \) is finite.\(^2\)

\(^2\)Notice that the absence of recursion in the patterns plays a fundamental role here: the set of possible traces has a finite dimension. Obviously, Proposition 3.1 holds up to structural equivalence, in particular as far as \( \alpha \)-renaming is concerned, since it causes infinite branching.
3.4. Correctness of Sessions

3.4.1 Correctness of closed sessions

Two notions of correctness are presented. The first one, more demanding, requires that a “successful” state is reached in every trace of a session, the second one, instead, requires that a successful state can be reached in at least one trace of the session. Even if traditional correctness as absence of deadlock requires correctness for all the traces, the weaker notion of “existential” correctness finds its motivation in those cases in which one wishes to check if at least a possibility exists that the system reaches a successful state. For example, in Chapter 5, an attacker for a security protocol is seen as a component that is able to drive the protocol to a successful termination towards at least a state where a security property is violated. A session is successful if all its patterns have been reduced to the empty behaviour 0, successfully completing their declared tasks.

Definition 3.5 (Successful session) Let \( S = \{(\bar{X}_1)[P_1], \ldots, (\bar{X}_n)[P_n]\} \) be a session. It is successful if and only if \( \forall i \in [1, \ldots, n] : P_i \equiv 0 \), (being \( \equiv \) structural equivalence).

A successful session may not necessarily be closed. This can happen, as illustrated in Example 3.6, when interaction patterns offer more open variables than they need to reach successful termination, for example because they can non-deterministically offer different behaviours, and hence channels, that are discarded by global choices.

Definition 3.5, adopted in this thesis, can be relaxed in order to study other properties than termination. For example a state could be considered successful when some check-points have been reached or when only the components that are critical for the correct functioning of the system have terminated. As in operating systems there are different access permissions to the machine functionality (often called application and kernel mode), it might be sensible to divide the components of a distributed application into two or more classes, with different capabilities, according to their role in the system. For instance, following [BBFT01], in security domains analysis the critical components, whose successful termination is required, are those implementing security protocols. This suggests possible ways of specialising the framework we are presenting by introducing a structure over the flat notion of session by means of a hierarchy of components, when required by the system under analysis.

A closed session, of whom all traces lead to a successful state is called (totally) correct:

Definition 3.6 (Totally correct session) Given a closed session \( S \), it is totally correct if and only if:

\[
\forall R : S \xrightarrow{\star} R \land R \not\xrightarrow{\perp} \Rightarrow R \text{ is successful}
\]

Example 3.7 Let us consider the traces of the two sessions of Example 3.4. The first session \( S_1 \), has been obtained by composing together a server and a client both
capable of managing the time-out message break. As expected, it is a correct session, since both its two different traces lead to a successful session:

\[
S_1 \xrightarrow{\text{out}(n,\text{query}), \text{in}(n,\text{Query})} S_2 \xrightarrow{\text{in}(n,\text{Answer}), \text{out}(n,\text{answer})} \{0\}, \{0\}
\]

\[
S_1 \xrightarrow{\text{out}(n,\text{query}), \text{in}(n,\text{Query})} S_2 \xrightarrow{(\tau)} S_3 \xrightarrow{\text{out}(n,\text{break}), \text{in}(n,\text{break})} \{0\}, \{0\}
\]

Let us consider now the construction of a session with the pattern

\[
\text{CLIENT} = [\text{out}(S,\text{query}). \text{in}(S,\text{Answer})]
\]

and the server capable of handling a break message:

\[
\{ (0) \left[ \text{out}(n,\text{query}). \text{in}(n,\text{Answer}).0 \right], \\
(0) \left[ \text{in}(n,\text{Query}). \left( \text{out}(n,\text{answer}).0 + \text{in}(n,\text{break}).0 \right) \right] \}
\]

The session results correct again, since its only possible trace leads to a successful session. Note how the global choice of the server, that accepts the input or the output in dependence of its environment, combined together with the pattern of the client let the session have only one successful trace.

Finally, let us consider the second session of Example 3.4, with a client sending a break message and a server that is not able to handle it. After the first communication, in case of the autonomous choice due to the \(\tau\) action of the client, the session reaches the following state:

\[
\{ (0) \left[ \text{out}(n,\text{break}).0 \right], (0) \left[ \text{out}(n,\text{answer}).0 \right] \}
\]

No action is possible nor is the state successful: The original session was not a (totally) correct session. Both the components try to output a value along a channel whose name is local to them, since it is not known outside the two patterns. Note that new patterns joining the session can not acquire the name \(n\), since they can not establish communication channels with the two components that know it. The session is permanently deadlocked, despite any new pattern joining it.

The notion of total correctness may appear too demanding requiring that all the possible traces of a session lead to a successful state and in some cases one would like to verify a weaker property.

We already mentioned the perspective of adding a structure to the flat sessions, so that one may wish to verify the total correctness of patterns belonging to some classes only.

Another obvious way of relaxing Definition 3.6 is to require the existence of a trace in which the session reaches a successful state. We hence introduce a (dual) notion of may-be correctness, or existential correctness.

**Definition 3.7 (May-be correct session)** Given a closed session \(S\), it is may-be correct if and only if:

\[
\exists \mathcal{R} : S \xrightarrow{^*} \mathcal{R} \land \mathcal{R} \text{ is successful}
\]
3.4. CORRECTNESS OF SESSIONS

The total or may-be correctness of a session can be verified by simply running the session itself and by analysing all its traces. According to Proposition 3.1, the verification of the correctness of a closed session amounts to exploring a finite set of states.

In [BBFT01], may-be correctness has been used to verify the safeness of security protocols. If, in one of the possible sessions where the protocol can be executed, a trace exists in which the protocol is broken, or equivalently (dually), it is not true that the protocol works properly in all the traces of the sessions where it may be executed, then it can not be considered safe. On the same approach is based the framework presented in Chapter 5.

3.4.2 Correctness of open sessions

Open systems appear as more complex systems to be analysed due to their partially “unpredictable” future behaviour, which depends on the behaviour of the components that will participate in the system and that are at the moment unknown. Necessarily, as also discussed in more details in Section 2.2.3, only weaker assessments about “correctness” can be stated.

The kind of analysis we propose permits us to verify local properties about the present behaviour of the system. In particular we are interested in checking if the interaction patterns in a session guarantee the “potential” (total or may-be) correctness of the up to now constructed system. In other words, given a session, we want to verify if it may be appropriately completed, so as to result correct. Moreover, we want to be able to check compatibility of an interaction pattern with a running session, in the sense that the access of the pattern will preserve the “potential” correctness of the session.

The weaker notion of correctness for open system we present is called acceptability. Intuitively, an open session is acceptable if an interaction pattern exists that can join the session making it closed and totally correct.

Definition 3.8 (Acceptable session) A session \( S \), where the set \( X_S \) is the union of the open variables of its patterns, is acceptable if and only if an interaction pattern \( (Y)[E] \) exists, disjoint from \( S \), and a mapping \( \gamma \) from \( (X_S \cup Y) \) to a set of fresh names such that:

\[
S' = \text{join}((Y)[E], \gamma, S)
\]

is a totally correct session. The pattern \( (Y)[E] \) is called a completion for \( S \).

The above definition reduces the acceptability of open sessions to the correctness of closed ones. Notice that the condition \( \text{dom}(\gamma) = (X_S \cup Y) \) implies that \( S' \) is a closed session: all the open variables have been bound. In this sense, the interaction pattern \( (Y)[E] \) represents, informally speaking, all the interaction necessary to the patterns already in the session for completing their coordinated tasks. In the future
evolutions of the session, the interaction expected by the patterns in the session may actually be provided by more components joining the session at different instants and possibly introducing new communication actions and new open variables that do not appear in \((Y)[E]\).

**Example 3.8 (Not acceptable session)** The open session \(S\), consisting of the components \(C_1, C_2\) and ORB of Example 3.5, connected in the intended way (by the mapping \([ch1/R1, ch1/Q1, ch3/S2, ch3/R2]\)):

\[
\{ 
\{(), [in(ch1,REQ1).0]\}, 
\{(), [out(ch3, ch2).in(ch2, REQ2).0]\}, 
\{C, [(in(C, Request). \\
\quad (\tau. out(C, ch1).0 + \tau.in(ch3, CH2).out(C, CH2).0) ) ] } 
\]

is not acceptable. Let us assume the existence of a completion \((Z)[E]\), with a corresponding mapping \(\gamma\), with \(\gamma(C) = c\) say, and let us show that a trace of \(S' = \text{join}((Z)[E], \gamma, S)\) exists which does not lead to a successful session.

This example also introduces the problem of how to construct a completion. In the trace, component \(E\) is not specified, and we assume it can only communicate through the channels it knows. With \(E_{(c)}\) we indicate that \(E\) knows the channel name \(c\), while \(\alpha_E\) stands for an action performed by \(E\):

\[
\{ 
\{(), [E_{(c)}]\}, 
\{(), [in(ch1, REQ1).0]\}, 
\{(), [out(ch3, ch2).in(ch2, REQ2).0]\}, 
\{(), [(in(C, Request). \\
\quad (\tau. out(C, ch1).0 + \tau.in(ch3, CH2).out(c, CH2).0) ) ] } 
\]

\(\overset{\text{in}(c, Request)}{\mapsto}\overset{\text{out}(c, request)}{\mapsto}\) \(E\)

\[
\{ 
\{(), [E_{(c)}]\}, 
\{(), [in(ch1, REQ1).0]\}, 
\{(), [out(ch3, ch2).in(ch2, REQ2).0]\}, 
\{(), [\tau. out(c, ch1).0 + \tau.in(ch3, CH2).out(c, CH2).0) ] } 
\]

\(\overset{\tau}{\mapsto}\)

\[
\{ 
\{(), [E_{(c)}]\}, 
\{(), [in(ch1, REQ1).0]\}, 
\{(), [out(ch3, ch2).in(ch2, REQ2).0]\}, 
\{(), [out(c, ch1).0] \}
\]

\(\overset{\text{out}(c, ch1), \text{in}(c, X)}{\mapsto}\) \(R\)
Now, only the patterns C2, the third one in the last session, knows the channel ch3, on which it is trying to send the channel name ch2. Since neither E nor any other pattern joining the session can access ch3, the pattern C2 can not successfully terminate. This example highlights the importance of a proper name sharing for patterns interacting with each other.

Example 3.9 (Acceptable session) In the previous example, acceptability of a session has been spoiled by a pattern offering a service that was not actually accessible in the session. More generally, the design of the patterns C1 and C2 is not coherent with the approach adopted, since C1 or C2 will continue to be present in the session indefinitely beyond the life of the patterns with which they were interacting.

The problem is overcome by redesigning ORB, C1 and C2 so that ORB lets the patterns whose services are not requested anymore successfully terminate:

\[
\begin{align*}
C1 &= (Q1) \left[ \text{in}(Q1, \text{REQ1}).0 + \text{in}(Q1, \text{terminate}).0 \right] \\
C2 &= (Q2) \left[ \text{in}(Q2, \text{REQ2}).0 + \text{in}(Q2, \text{terminate}).0 \right] \\
ORB &= (C, R1, R2) \left[ \text{in}(C, \text{Request}). \right. \\
&\quad \left. (\tau.\text{out}(C, R1).\text{out}(R2, \text{terminate}).0 + \tau.\text{out}(C, R2).\text{out}(R1, \text{terminate}).0) \right] \\
\end{align*}
\]

By means of the mapping [ch1/Q1, ch1/R1, ch2/Q2, ch2/R2] we then obtain the following open session:

\[
\begin{align*}
\{ & \left[ \text{in}(ch1, \text{REQ1}).0 + \text{in}(ch1, \text{terminate}).0 \right], \\
& \left[ \text{in}(ch2, \text{REQ2}).0 + \text{in}(ch2, \text{terminate}).0 \right], \\
& \left[ \text{in}(C, \text{Request}). \right. \\
&\quad \left. (\tau.\text{out}(C, ch1).\text{out}(ch2, \text{terminate}).0 + \tau.\text{out}(C, ch2).\text{out}(ch1, \text{terminate}).0) \right] \} \\
\end{align*}
\]

which is acceptable. The pattern REQUEST:

\[
\begin{align*}
\text{REQUEST} &= (S)[\text{out}(S, \text{request}).\text{in}(S, \text{CH}).\text{out}(\text{CH}, \text{call_service}).0] \\
\end{align*}
\]

\footnote{This, obviously, under the assumption that ch3 is not a global name.}
by the mapping \([\text{ch3/S, ch3/C}]\), is a completion for it.

It is now possible to specify the rule \((\text{join})\) according to the introduced notion of acceptability.

**Definition 3.9 \((\text{in})\) rule** Let \(S\) be an acceptable session with \(X_S\) the union of the open variables of the interaction patterns in the session. Let \((Y)[E]\) be an interaction pattern, and let \(\gamma\) be a mapping from \(V \subseteq (X_S \cup Y_E)\) to a set of fresh names. The following rule \((\text{in})\) completes the semantics of sessions by instantiating rule \((\text{join})\):

\[
S' = \text{join}(Y)[E], \gamma, S) \quad S' \text{ is acceptable}
\]

\[
S \xrightarrow{\text{(in)}} S'
\]

The correctness of closed systems can be investigated by means of a finite amount of information, while about open systems, by means of a finite process, we can only check for acceptability. Acceptability regards the possible traces of an “isolated” session, to which rule \((\text{join})\) can not be applied. Clearly, while \((\text{join})\) can in principle spoil acceptability of a session, conversely rules \((\text{comm})\), \((\text{silent})\) and \((\text{in})\) preserve it. By using \((\text{in})\) in place of \((\text{join})\), acceptability becomes an invariant property that holds throughout the potentially infinite life of a session continuously joined by “good” interaction patterns.

**Theorem 3.1 (Subject reduction)** Let \(S\) be an acceptable session. If \(S': S \xrightarrow{\text{comm}} S'\) by rule \((\text{comm})\), \((\text{silent})\) and \((\text{in})\) then \(S'\) is acceptable.

**Proof.**

If \(S': S \xrightarrow{\text{in}} S'\) by rule \((\text{in})\), then \(S'\) is acceptable by definition of \((\text{in})\). On the other hand (see Figure 3.6), let \(P\) (with the respective mapping \(\gamma\)) be a completion for \(S\), and \(S': S \xrightarrow{\text{comm}} S'\) by rule \((\text{comm})\) or \((\text{silent})\). Then, since the open variables of \(S'\) are the same as those of \(S\), we can define \(R' = \text{join}(P, \gamma, S')\).

From \(R = \text{join}(P, \gamma, S)\) there is a transition, i.e. the one which applies exactly the same rule (to the same patterns) used in \(S \xrightarrow{\text{comm}} S'\), such that \(R \xrightarrow{\text{comm}} R'\). Since \(R\), by hypothesis, is a totally correct closed session, also \(R'\) is a closed and totally correct session. It follows that \(P\) is a completion for \(S'\), that is hence acceptable.
3.5 Verifying acceptability

Practical aspects of the proposed methodology rely on the verification of acceptability. This section presents an algorithmic approach to construct a completion for a given session, if there is any.

The algorithm tries to construct one of the possible completions by non-deterministically exploring the finite state space of the traces of the session, and it is proved to be correct: if it returns not acceptable then the session is not acceptable, while if it returns an interaction pattern, then it is a completion for the session. The algorithm always terminates.

The construction of the candidate completion proceeds incrementally. Some care is necessary to manage the effects of extending the completion with a new action, since adding an action can make the traces of the session, to which the completion is joined, vary dramatically. The key feature of our procedure is that it exhaustively explores the effects of the last action added, before extending the completion further.

In order to simplify the construction, only completions having a particular structure, called standard form, are attempted to be constructed. After introducing the standard form, it is proved that this does not cause loss of generality, since if a generic completion exists, then a standard form completion exists, too.

Then, the algorithm is illustrated and its complexity discussed. Some examples of the algorithm at work are provided in order to facilitate its understanding. Finally, the correctness of the algorithm is proved.

In this section we use a slightly different representation of session semantics (Figure 3.5), in order to distinguish the transitions in which the under-construction completion participates, from those in which the session under analysis autonomously evolves. We write \( \langle S, P \rangle \xrightarrow{\alpha} \langle S', P' \rangle \) as a shorthand for a transition in which the interaction pattern \( P \), i.e. a possible completion, communicates with one of the interaction patterns in \( S \) by means of the action \( \alpha \). Note that the meaning of the label \( "-\alpha" \) is that the actions done by \( P \) can be distinguished from the others in a trace. More precisely, \( S = R \cup \{Q\} \) and the transition is done according to either a rule like

\[
\begin{array}{c}
P \xrightarrow{\text{in}(c,d')} P' \\
Q \xrightarrow{\text{out}(c,d)} Q' \\
\exists \sigma, d'\sigma = d
\end{array}
\]

\[
\begin{array}{c}
R \cup \{Q\} \cup \{P\} \\
\xrightarrow{\text{out}(c,d),\text{in}(c,d')} \\
R \cup \{Q'\} \cup \{P''\sigma\}
\end{array}
\]

in which \( P \) performs an input action (with \( P' = P''\sigma \) and \( S' = R \cup \{Q'\} \)), and \( \alpha = \text{in}(c,d') \), or a rule like

\[
\begin{array}{c}
P \xrightarrow{\text{out}(c,d)} P' \\
Q \xrightarrow{\text{in}(c,d')} Q' \\
\exists \sigma, d'\sigma = d
\end{array}
\]

\[
\begin{array}{c}
R \cup \{Q\}, \cup \{P\} \\
\xrightarrow{\text{in}(c,d'),\text{out}(c,d)} \\
R \cup \{Q'\sigma\}, \cup \{P'\}
\end{array}
\]

in which \( P \) receives a datum (and \( S = R \cup Q'\sigma \)), and \( \alpha = \text{out}(c,d) \).
Analogously, for a transition in which the session $S$ evolves independently of $P$, we write either $\langle S, P \rangle \xrightarrow{\alpha \bar{\alpha}} \langle S', P \rangle$, in case that $Q$ and $R$, which are in $S$, communicate to each other by means of the dual actions $\alpha$ and $\bar{\alpha}$. Precisely, $S \cup \{P\} = R \cup \{Q', R\} \cup \{P\} = S' \cup \{P\}$. Otherwise, we write $\langle S, P \rangle \xrightarrow{\tau} \langle S', P \rangle$, when one of the interaction patterns in $S$ performs a silent action.

Finally, $\langle S, P \rangle \xrightarrow{\tau_{\Lambda}} \langle S', P' \rangle$, stands for a transition in which the completion $P$ evolves independently of $S$ by means of a silent action. About the last two cases, we assume again that it is possible to understand if $\tau$ has been performed by $P$ (last case) or not (last but one case).

Obviously, in searching for completions, transitions due to rule (in) are not of interest. As before, $\xrightarrow{+}$, $\xrightarrow{\ast}$ and $\not\xrightarrow{}$ have the standard meaning, while in $\xrightarrow{\Lambda}$ the actions executed in the trace are indicated by the string of labels $\Lambda$.

### 3.5.1 Completion standard form

In order to make completion construction easier, it has been restricted to the construction of completions in **standard form**.

It will be proved that the existence of a completion implies the existence of a standard form completion. Since acceptability is implied by the existence of a completion, searching for a standard form completion is not less general than searching for a generic completion when verifying acceptability.

Informally speaking, the behavioural expression of a standard form completion does not contain the parallel composition operator $||$, nor silent actions. About the former, it is a well known result that, under appropriate hypothesis, an interaction pattern $(X)[E]$ can be transformed in a strongly bisimilar (Section 1.5.2) expression of the interaction pattern $(X)[\mathcal{F}(P)]$, which does not contain the $||$ operator. The transformation is analogous to that of CCS normal form [Mil89], considering, in particular that sub-terms of an interaction patterns do not communicate with each other. Briefly, the expression $E||F$ can be recursively transformed in the expression:

$$\mathcal{F}(E||F) = \alpha_1.\mathcal{F}(E_1||F) + \ldots + \alpha_n.\mathcal{F}(E_n||F) + \beta_1.\mathcal{F}(E||F_1) + \ldots + \beta_m.\mathcal{F}(E||F_m),$$

where $\forall i \in [1..n] E \xrightarrow{\alpha_i} E_i$ and $\forall j \in [1..m] F \xrightarrow{\beta_j} F_j$.

About silent actions, it appears reasonably true that a completion does not need to impose local choices to the other patterns in the session, and hence, as will be proved, if a completion exists, then a completion in which silent actions do not occur also exists.

**Definition 3.10 (Standard form)** An interaction pattern $(X)[E]$ is in **standard form** if its behavioural expression $E$ is in standard form, i.e. it is generated by the following grammar:

\[
\begin{align*}
SF & ::= F | F + SF \\
F & ::= \alpha | \alpha.SF
\end{align*}
\]
where \( \alpha \) is a communication action.

Behavioural expressions in standard form can be represented as a tree, where branching corresponds to +, while child relation corresponds to prefix, and the root of the tree is represented by a distinguished label \( r \) (root). The behaviour of a normal form interaction pattern \( P \) in a trace corresponds to a path in the tree. Hence, the standard form expression \( \alpha + (\beta.(\gamma + \alpha)) \) corresponds to the tree:

```
    r
   /\  \
  /  \  /
 α  β \\
 /\  /\  /
| | | |  |
γ α α
```

In the following, we assume that it is possible to univocally refer a occurrence of an action in a behavioural expression, so that, for instance, it will clear from the context which of the two \( \alpha \) actions in the tree we are speaking about (this can be achieved, for example, by simply associating any visiting order to the tree).

Given a completion \( P = (X)[E] \) where the \( \parallel \) operator does not occur, we indicate with \( \hat{P} = (X)[\hat{E}] \) a \( \tau \)-less “equivalent” completion, defined as follows. Unfortunately, \( \tau \) action can not be simply deleted. Consider, for example, a completion with an expression \( E = \tau.0 + \alpha.0 \). Simply deleting the \( \tau \) would yield the expression \( \hat{E} = 0 + \alpha.0 \equiv \alpha.0 \). Suppose that \( \alpha \) is not a necessary action to drive the session through a successful trace (for example because the session is already successful). While \( E \) has the choice of “doing nothing”, \( \hat{E} \) is forced to synchronise on \( \alpha \) together with the session, even if the session can not.

The following definition of \( \hat{E} \) overcomes this problem by checking the possibility for \( E \) of “doing nothing”. Note that the property \( E \xrightarrow{\tau \ast} 0 \), i.e. “it exists a trace in which \( E \) rewrites to 0 in an arbitrary number of \( \tau \) actions”, is decidable.

**Definition 3.11** Let \( P = (X)[E] \) be an interaction pattern in which \( \parallel \) does not occur. Then the interaction pattern \( \hat{P} = (X)[\hat{E}] \) is defined as follows:

\[
\hat{E} = \begin{cases} 
0 & \text{if } E \xrightarrow{\tau \ast} 0 \\
\hat{F} & \text{if } E = \tau.F \land \neg(E \xrightarrow{\tau \ast} 0) \\
\alpha.\hat{F} & \text{if } E = \alpha.F \\
\hat{F} + \hat{G} & \text{if } E = F + G \land \neg(E \xrightarrow{\tau \ast} 0)
\end{cases}
\]

Lemma 3.1 states the property of the construction “−” in terms of bisimulation between \( P \) and \( \hat{P} \) up to \( \tau \) actions (weak bisimulation).

**Lemma 3.1** Let \( P \) be an interaction pattern in which the parallel operator \( \parallel \) does not occur. Then it holds:
1. If $\hat{P} \xrightarrow{\alpha} \hat{P}'$ then $P \xrightarrow{\tau} \xrightarrow{\alpha} P'$.

2. If does not exist a trace for $P$ such that $P \xrightarrow{\tau} 0^4$ then, if $P \xrightarrow{\tau} \xrightarrow{\alpha} P'$ then $\hat{P} \xrightarrow{\alpha} \hat{P}'$.

Proof.

(proof of 1)$^5$
If $P \xrightarrow{\tau} 0$ then $\hat{P} = ()[0]$ and the implication vacuously holds.

Otherwise, the thesis is proved by structural induction on $P$. The base case, i.e. $P = ()[0]$, falls in the previous case.

If $P = ()[\tau.E]$ then $\hat{P} = ()[\hat{E}]$, and hence, if

$$()[\hat{E}] \xrightarrow{\alpha} ()[\hat{E}'],$$

then

$$()[\tau.E] \xrightarrow{\tau} ()[E] \xrightarrow{\tau} \xrightarrow{\alpha} ()[E'].$$

By inductive hypothesis.

If $P = ()[\alpha.E]$ then $\hat{P} = ()[\alpha.\hat{E}]$, and hence, if

$$()[\alpha.\hat{E}] \xrightarrow{\alpha} ()[\hat{E}],$$

then

$$()[\alpha.E] \xrightarrow{\alpha} ()[E].$$

If $P = ()[F + G]$ then $\hat{P} = ()[\hat{F} + \hat{G}]$, and hence $\hat{P}$ can evolve according to either $\hat{F}$ or $\hat{G}$. Let us suppose it evolves according to $\hat{F}$ (the case for $\hat{G}$ is symmetric). If

$$()[\hat{F} + \hat{G}] \xrightarrow{\alpha} ()[\hat{F}'],$$

then $()[\hat{F}] \xrightarrow{\alpha} ()[\hat{F}']$, and, by inductive hypothesis, $()[F] \xrightarrow{\tau} \xrightarrow{\alpha} ()[F']$, and hence

$$()[F + G] \xrightarrow{\tau} \xrightarrow{\alpha} ()[F'].$$

(proof of 2)
By structural induction on $P$. If $P = ()[0]$, than $\hat{P} = ()[0]$, and the thesis holds.

If $P = ()[\tau.E]$ then, by hypothesis on $P$,

$$()[\tau.E] \xrightarrow{\tau} ()[E] \xrightarrow{\tau} \xrightarrow{\alpha} ()[E'],$$

$^4$With a small abuse of notation 0 is here an interaction pattern whose behavioural expression is 0.

$^5$In this proof (and in the following, whenever useful for the ease of presentation), open variables, when not significant, have been omitted.
and hence, by induction:

\[ \hat{P} = (\hat{E}) \mathrel{\alpha} (\hat{E}). \]

by inductive hypothesis

If \( P = (\alpha.E) \) then

\[ (\alpha.E) \mathrel{\alpha} (E), \]

and,

\[ \hat{P} = (\alpha.\hat{E}) \mathrel{\alpha} (\hat{E}). \]

Finally, if \( P = (F+G) \), it can evolve according to either \( F \) or \( G \). Let us suppose \( P \) evolves according to \( F \) (the case for \( G \) is symmetric), which must eventually perform an action \( \alpha \neq \tau \):

\[ (F) \mathrel{\tau^*} (F'). \]

Then,

\[ (F+G) \mathrel{\tau^*} (F'). \]

By the hypothesis on \( P \), \( \hat{P} = (\hat{F}+\hat{G}) \), and, by inductive hypothesis, \( (\hat{F}) \mathrel{\alpha} (\hat{F}) \). It follows that

\[ (\hat{F}+\hat{G}) \mathrel{\alpha} (\hat{F}). \]

\[ \square \]

Clearly, one would expect that if \( P \) is a completion for a session, then also \( \hat{P} \) is. The bisimulation relation holding between \( P \) and \( \hat{P} \) scales up to sessions, and it is possible to prove that, given a session \( R, S = \text{join}(P, \gamma, R) \) and \( \hat{S} = \text{join}(\hat{P}, \gamma, R) \) enjoy a similar simulation property: all the transitions of \( \hat{S} \) can be simulated by corresponding transitions of \( S \). It follows that a possible deadlock in \( \hat{S} \), preventing \( \hat{P} \) to be a completion, must have a corresponding deadlock in \( S \), which is not consistent with the hypothesis of \( P \) being a completion.

The property of simulation is firstly defined in general as a relation among sessions, and then it is shown that the pair \( (\hat{S}, S) \) fulfills the definition.

**Definition 3.12 (Session weak simulation)** A binary relation over sessions is a session weak simulation, written \( \sqsubseteq_w \), if and only if, \( S_1 \sqsubseteq_w S_2 \) implies

\[ i) \quad \exists S' \quad S_1 \mathrel{\tau} S' \quad \Rightarrow \quad \exists S'_2 \quad S_2 \mathrel{\tau} S'_2 \quad \land \quad S'_1 \sqsubseteq_w S'_2, \quad \text{and} \]

\[ ii) \quad \forall \alpha. \quad \exists S' \quad S_1 \mathrel{\alpha} S'_1 \quad \Rightarrow \quad \exists S'_2 \quad S_2 \mathrel{\tau^* \alpha} S'_2 \quad \land \quad S'_1 \sqsubseteq_w S'_2. \]

To prove that \( \hat{S} \sqsubseteq_w S \) let us show that the relation

\[ T = \{((\langle R, \hat{P} \rangle, \langle R, P \rangle)) \} \]

is a session weak simulation.
Theorem 3.2 Let be $\langle R, \hat{P}\rangle$ and $\langle R, P\rangle$ two sessions which differ for $\hat{P}$ being the normal form of $P$, then the relation $T = \{ (\langle R, \hat{P}\rangle, \langle R, P\rangle) \}$ is a session weak simulation.

Proof.

About case i) of Definition 3.12, we note that, since no $\tau$ occurs in $\hat{P}$,

$\langle R, \hat{P}\rangle \xrightarrow{\tau} \langle R', \hat{P}\rangle$

only if $R$ performs a $\tau$, but then, by the same transition

$\langle R, P\rangle \xrightarrow{\tau} \langle R', P\rangle$

and, by definition of $T$, $\langle R', \hat{P}\rangle, \langle R', P\rangle \in T$.

About case ii), we need to prove that,

$\forall \alpha. \langle R, \hat{P}\rangle \xrightarrow{\alpha} \langle R', \hat{P}'\rangle \Rightarrow \exists Q. \langle R, P\rangle \xrightarrow{\tau, \alpha} \langle R', Q\rangle \land \hat{Q} = \hat{P}$

If $P \xrightarrow{\tau, *} 0$ then $\hat{P} = ()[0]$, and hence $\langle R, 0\rangle \xrightarrow{\alpha, \alpha} \langle R', 0\rangle$. Then, with the same transition involving $R$, also $\langle R, P\rangle \xrightarrow{\alpha} \langle R', P\rangle$, and $\langle (R', 0), (R', P) \rangle \in T$.

The case in which $\neg (P \xrightarrow{\tau, *} 0)$ is proved by structural induction on $P$. Note that the base case of induction, $P = ()[0]$, holds because of the previous point.

If $P = ()[\tau.E]$, then $\hat{P} = ()[\hat{E}]$. If

$\langle R, ()[\hat{E}]\rangle \xrightarrow{\alpha} \langle R', ()[\hat{E}]\rangle$. then

$\langle R, ()[\tau.E]\rangle \xrightarrow{\tau} \langle R, ()[E]\rangle \xrightarrow{\alpha, \alpha} \langle R', ()[E']\rangle$. by inductive hypothesis

and the pair $\langle (R', ()[\hat{E}']), (R', ()[E']) \rangle$ belongs to $T$.

If $P = ()[\beta.E]$, then $\hat{P} = ()[\hat{E}]. If$

$\langle R, ()[\hat{E}]\rangle \xrightarrow{\alpha} \langle R', ()[\hat{E}]\rangle$,

then

$\langle R, ()[\beta.E]\rangle \xrightarrow{\alpha, \alpha} \langle R', P\rangle$.

In fact, if $\beta = \hat{\alpha}$ then $P' = ()[E]$ and $\hat{P}' = ()[\hat{E}]$, if, otherwise, $\beta \neq \hat{\alpha}$ then $P' = ()[\beta.E]$ and $\hat{P}' = ()[\hat{\beta.E}$. In both the cases, the pair $\langle (R', \hat{P}'), (R', P') \rangle$ belongs to $T$. 
3.5. VERIFYING ACCEPTABILITY

Finally, if \( P = (F + G) \), then \( \hat{P} = (\hat{F} + \hat{G}) \). Let us suppose that

\[
\langle \mathcal{R}, (\hat{F} + \hat{G}) \rangle \xrightarrow{\lambda} \langle \mathcal{R}', \hat{P}' \rangle,
\]

with \( \lambda \) one of the possible labels.

Three cases are possible:

1. \( \hat{P}' = (\hat{F} + \hat{G}) \). Then \( \mathcal{R} \) “moved alone” and \( \lambda = \alpha\bar{\alpha} \). Then the following transition from \( \langle \mathcal{R}, P \rangle \) exists, (where \( P' = P \)):

\[
\langle \mathcal{R}, P \rangle \xrightarrow{\lambda} \langle \mathcal{R}', P' \rangle.
\]

2. \( \hat{P}' = (\hat{F}') \). The transition happened because of the transition of the interaction pattern \( \hat{P} = (\hat{F} + \hat{G}) \xrightarrow{\alpha} \hat{F}' \), and \( \lambda = -\alpha \). By Lemma 3.1, \( P = (F + G) \xrightarrow{\tau^*} F' \), and then

\[
\langle \mathcal{R}, (F + G) \rangle \xrightarrow{\tau^* \lambda} \langle \mathcal{R}', (F') \rangle.
\]

3. \( \hat{P}' = (\hat{G}') \). Symmetric to case 2.

For each one of the previous cases, it holds \( (\langle \mathcal{R}', \hat{P}' \rangle, \langle \mathcal{R}', P' \rangle) \in T \). It follows that \( T \) is a session weak simulation \( (\langle \mathcal{R}, \hat{P} \rangle \sqsubseteq_w \langle \mathcal{R}, P \rangle) \).

\[\square\]

By means of Lemma 3.1, Theorem 3.2 proved that all the transitions of a session \( \hat{S} \) “can be simulated” by the corresponding session \( S \). The repeated application of this result permits us to state that every trace of \( \hat{S} \) has a correspondent in a trace of \( S \). By means of this observation, the following theorem states that if \( P \) is a completion for a session \( \mathcal{R} \) then also \( \hat{P} \) is a completion for the same session.

**Theorem 3.3** Let \( \mathcal{R} \) be a session, \( \gamma \) a mapping and \( P \) an interaction pattern such that \( S = \text{join}(P, \gamma, \mathcal{R}) \) is a closed and totally correct session (\( P \) is a completion for \( \mathcal{R} \)). Then \( \hat{S} = \text{join}(\hat{P}, \gamma, \mathcal{R}) \) is a closed and totally correct session (\( \hat{P} \) is a completion for \( \mathcal{R} \)).

**Proof.**

(by absurdum)

Obviously \( \hat{S} \) is a closed session. Let us suppose that \( \hat{S} \) is not totally correct, i.e.

\[
\exists \hat{S}', \hat{S}' \xrightarrow{\tau^*} \hat{S}' = \langle \mathcal{R}', \hat{P}' \rangle \not\in \hat{S}' \text{ is not successful.}
\]

By repeatedly applying the fact that \( \hat{S} \sqsubseteq_w S \), it exists a trace \( S \xrightarrow{\tau^*} S' = \langle \mathcal{R}', P' \rangle \). The session \( \langle \mathcal{R}', P' \rangle \) is not successful, otherwise also \( \hat{S}' = \langle \mathcal{R}', \hat{P}' \rangle \) would
be successful. Hence, since $S$ is totally correct ($P$ is a completion for $R$) and $S'$ is not successful, there exist $R''$ and $P''$ such that
\[
\langle R', P' \rangle \not\rightarrow \langle R'', P'' \rangle.
\]
Let us suppose that $R'$ is a successful session, then $P'$, in order to let the whole session $\langle R', P' \rangle$ successfully terminate, must autonomously reduce to 0, i.e. $P' \not\rightarrow^* 0$. In this case $\hat{P}' = 0$, and then $\langle R', \hat{P}' \rangle$ is a successful, too, against the hypothesis.

On the other hand, $R'$ can not move alone, $R' \not\rightarrow^*$, because $\langle R', \hat{P}' \rangle$ is a deadlock state.

Since $R' \not\rightarrow^*$ and $P \not\rightarrow^* 0$, necessarily $\exists \alpha. P' \not\rightarrow^* \alpha \rightarrow^* P''$, and $\langle R', P' \rangle \not\rightarrow^* \langle R'', P'' \rangle$.

By Lemma 3.1, $\hat{P}' \not\rightarrow^* \hat{P}''$, and $\langle R', \hat{P}' \rangle \not\rightarrow^* \langle R'', \hat{P}'' \rangle$, against the hypothesis of being a deadlock state.

By Theorem 3.3, it follows that in checking the acceptability of a session, i.e. verifying the existence of a completion, it is safe to restrict the search to completions in standard form.

3.5.2 The algorithm $A$ (Acceptability checker)

The algorithmic construction of completions presents some difficulties due to the non-compositional nature of the problem. Consider, for instance, a session $\{()[\alpha.0 + \alpha.\beta.0]\}$ consisting of one interaction pattern.

One could try to address the problem by structurally decomposing the session, and searching for partial completions for the two non-deterministic alternatives of the interaction pattern.

Unfortunately, putting together the successfully constructed “partial completions” $\bar{\alpha}.0$ and $\bar{\alpha}.\bar{\beta}.0$, is not straightforward. In fact, in the tentative completion $\{()[\bar{\alpha}.0 + \bar{\alpha}.\bar{\beta}.0]\}$, each one of the partial completions can interact with the whole session, so that $\bar{\alpha}.\bar{\beta}.0$ of the completion and $\alpha$ of the pattern can lead to the deadlock session $\{[0],[\beta.0]\}$.

The alternative approach we followed, consists in incrementally constructing the completion by expanding it with an action, called trigger, able to trigger a deadlock state (we present a non-deterministic specification of the algorithm, where in presence of more triggers, one of them is non-deterministically chosen). For each action added to the completion, the whole state of the traces of the session and the new completion is reconsidered and the effects of the last action added are evaluated, and possibly backtracked in case of failure.
3.5. VERIFYING ACCEPTABILITY

The not acceptable context \{[\alpha.\beta + \alpha]\}

Figure 3.7: Algorithm \(\mathcal{A}\) at work - I.

The procedure ends when either a completion is found by having added enough actions so as to eliminate all the deadlocks, or the session is declared not acceptable, since it is not possible to eliminate all the deadlocks.

To give the flavor of the algorithm functioning, let us follow its application to the previous simple example by means of a graphical notation.

In this notation, actions are indicated as usual as \(\alpha, \bar{\alpha}, \beta, \ldots\), while open variables are not taken into account, assuming that the completion has access to all the “open” channels and to those that it can acquire by interacting with the session.

Each step of the algorithm is graphically represented (see, for instance, Figure 3.7) as a box containing:

1. the session and the so far expanded completion, i.e. \(\{S, P\}\) – the last action of \(P\) is marked by an arrow (the empty behaviour \(0\) is omitted),

2. the tree of the reachable states from the session, and hence its traces, – deadlocked states are marked with a circle, and sometime the traces that are influenced by the last action added to the completion are shaded,

3. the conditions that justify the next step, like FAIL, RETURN, or the set of triggers – if more then one trigger is available, the next step(s) chooses nondeterministically one of them (i.e. the step has more then one child).
CHAPTER 3. INTERACTION PATTERNS AND SESSIONS

In Figure 3.7, the session \{[\alpha.\beta + \alpha]\} is checked for acceptability. The first box represents the initial state space of the traces for the session, containing the deadlocked session alone. Only the action \(\bar{\alpha}\) can be used as a trigger, and it is hence used to expand the completion. Since it interacts with the two choices of the pattern, two traces are generated, of which one is successful, and the other is deadlocked. The shaded traces (both of the two) are the traces influenced by the last modification of the under-construction completion.

Informally, the algorithm keeps on expanding the completion while the introduction of the triggering actions permits one or more deadlock states to be removed without introducing new ones. If all the deadlock states can be embraced in this process, finally a correct completion is returned.

In the example, since there is a trace, to which \(\alpha\) participates, that is still not successful, the completion ought to perform other triggering actions after \(\alpha\). Actually, it is not possible to further expand the completion, since any new action added “after” \(\alpha\) would spoil the other already successful trace in which \(\alpha\) participate, by introducing new deadlock states. FAIL is hence returned, since the session is not acceptable.

Other examples of the algorithm at work can be found in Section 3.5.2.

Before introducing and explaining the specification of the algorithm, it is worth discussing some issues on which the algorithm depends. Some of these issues are dealt with separately by a set of procedures that are called by the main body of algorithm. Sometimes, the procedures simply represent the computational counterpart of previously given definitions (like, e.g, for the case of deadlock states).

Reachable states

The space of all the possible traces is generated by the procedure \(\text{nodes}(\langle S, P \rangle)\), which returns the set of all the reachable states from \(\langle S, P \rangle\). This set will then be sought for deadlock states. Formally, according to the notation introduced on page 65:

\[
\text{nodes}(\langle S, P \rangle) = \{\langle R, Q \rangle \mid \langle S, P \rangle \rightarrow^* \langle R, Q \rangle\}.
\]

Deadlock and successful states

Within a set of states, it is important to distinguish those that are deadlocked. and those that are successful. Given \(N\), a set of states (of the form \(\langle R, P \rangle\)), the set of deadlocked nodes in \(N\) is returned by the procedure \(\text{deadlocks}(N)\):

\[
\text{deadlocks}(N) = \{\langle R, P \rangle \mid \langle R, P \rangle \in N \land \langle R, P \rangle \not\rightarrow^* \langle R, P \rangle \land \langle R \text{ not successful} \lor P \neq 0\}\},
\]

and the set of the successful nodes \(N\) is returned by the procedure \(\text{successes}(N)\):

\[
\text{successes}(N) = \{\langle R, P \rangle \mid \langle R, P \rangle \in N \land (R \text{ successful} \land P \equiv 0)\}\}.
\]
Completion knowledge

A completion, like any other interaction pattern, can communicate only through the channels of which it knows the name. Usually, an interaction pattern can acquire a name by means of an open variable, a communication, or it can know the name since it occurs in its behavioural expression. In the case of an under-construction completion, which has not (yet) a (completely) defined interaction pattern, the information about the channels it can access must be explicitly recorded.

With $\kappa$ we indicate the set of names that represents the (current) knowledge of the completion.

Initially, the completion, that must make the session closed and correct, must be connected to all the channels that are open in the session. Let $X_\mathcal{S}$ be the set of the open variables of $\mathcal{S}$, we assume that $P$ has as many open variables as $\mathcal{S}$, which we indicate with $X_P$ (a set of variables not occurring in $X_\mathcal{S}$). All the open variables in $X_P$ are connected one-to-one to those in $X_\mathcal{S}$, by an appropriate mapping $\gamma : X_\mathcal{S} \cup X_P \rightarrow \eta$, where $\eta$ is a set of fresh names. It follows that initially $\kappa$ must contain $\eta$.

As the construction of $P$ proceeds, $\kappa$ may grow by means of the names that $P$ may acquire from communications. Moreover, $P$ must be provided with the capability of generating a finite set of new names, when requested by its partners in the communications. This correspond, in the $\pi$-calculus jargon, to the extrusion of its local names, and is implemented, coherently with literature, by providing $P$ with a set of fresh names in $\kappa$. Note that a finite upper bound to the number of needed names can be given by the number of communications in the session under analysis.

The procedure $\text{update}(\kappa, \beta)$, is in charge both of updating $\kappa$ with the names that $P$ receives by means of an action $\beta$, and of keeping track of the names sent by an action $\beta$, so as to guarantee the possibility of choosing a fresh name when needed. In general, creation, managing and deallocation of names is difficult since it may easily lead to infinite branching semantics or it may be difficult to keep trace of the identity of names, see for example [Pis99]. In our specific case, the completion may need to communicate either through one of the known channels, or through a newly created channel. The relevant characteristic of the new channel is to be different from all the others in the session, and hence any new name can be indifferently used. In any case, the number of different channels accessed in a session can not be infinite.

In different scenarios, identity of names (as well as of application data) may become semantically significant, and it may be that a fresh name can not be simply characterised as a new one. For example, in the analysis of secure protocols presented in Chapter 5, the identity of infinitely many different keys and other “application data” may be significant in order to unveil a secret. In order to deal with an infinity of different names, more sophisticated name managing mechanisms, like symbolic analysis, will be introduced.
Triggering actions

The set of actions by means of which \( P \), with its current knowledge \( \kappa \), can trigger (at least) one deadlock state in a set \( N \), is returned by the procedure \( \text{triggers}(N, \kappa) \). Among the actions \( \alpha \), which an interaction pattern \( Q \) in the deadlock state is ready to perform, only those whose channel is known to \( P \), i.e. \( ch(\alpha) \in \kappa \), can be selected. Note that the procedure returns the dual of the action which \( Q \) is ready to perform, i.e., an action that can be used to expand \( P \). Formally:

\[
\text{triggers}(N, \kappa) = \{ \alpha | \exists (R, P) \in N, \exists Q, Q': (\cdot)[Q] \in R \land Q \xrightarrow{\alpha} Q' \land (R, P) \not\xrightarrow{\cdot} \land ch(\alpha) \in \kappa \}
\]

States affected by expanding the completion

As pointed out, a feature of the algorithm is to globally check the consequences of having expanded the under construction completion with a new action. As soon as the completion has been extended with an action \( \alpha \), all the states that can be reached through a trace in which \( \alpha \) is performed are checked. Necessarily, \( \alpha \) is the last action that \( P \) does, since no new actions have been yet added after it.

The procedure \( \text{evolutions}_\text{after}((S, P), \alpha) \) returns the set of states, among those reachable from \( (S, P) \), which can be reached from the session \( (S, P) \) by a trace in which \( \alpha \) occurs as the last action done by \( P \).

The case in which no action has been added to \( P \) (or, the root element of the normal form completion, see Section 3.5.1, is taken into consideration because of backtracking, see the algorithm specification) is covered by the case \( \text{evolutions}_\text{after}((S, P), \text{root}) \). It returns all the states that can be reached from the session \( (S, P) \) without the contribution of the completion. Formally:

\[
\text{evolutions}_\text{after}((S, P), \alpha) = \{ (R, Q) | (S, P) \xrightarrow{\Lambda} (R, Q) \land \Lambda = \{ (\cdot) \}, \{ (\cdot, \cdot) \}, \{ (\cdot, \cdot) \} \}
\]

\[
\text{evolutions}_\text{after}((S, P), \text{root}) = \{ (R, P) | (S, P) \xrightarrow{\Lambda} (R, P) \land \Lambda = \{ (\cdot, \cdot) \} \}
\]

Note \( \Lambda \) in the definition, i.e. the sequence of labels of a trace leading to an \( \text{evolution}_\text{after} \) state: if \( \alpha \) is an action of the completion, \( \Lambda \) consists of a generic sequence of actions of \( (S, P) \), followed by \( (\cdot) \), i.e. the transition in which \( P \) performs \( \alpha \), followed by actions that are not done by \( P \) (according to the notation introduced on page 65). If \( \alpha = \text{root} \), then \( \Lambda \) consists of a sequence of actions that \( S \) can autonomously perform (\( P \) does not evolve).

The algorithm \( \mathcal{A} \) (Acceptability checker), represented in Figure 3.8 is a non-deterministic algorithm. Non-determinism is expressed by the construct \( \text{CHOICE}(S) \), where \( S \) is a set. Typically \( \text{CHOICE} \) is used as a denotable value in an assignment.
\[ A(\mathcal{S}, \kappa) \]

\[ (\gamma, \eta) = \text{define}\_\text{mapping}(X_\mathcal{S}, X_P); \]

\[ \mathcal{S} = \mathcal{S}_{\gamma}; \]

\[ \kappa = \eta \cup \kappa; \]

\[ \text{if } \text{deadlocks}(\text{nodes}(\langle \mathcal{S}, \emptyset \rangle)) = \emptyset \]

\[ \text{then return([ ], \gamma)} \quad \text{// } S \text{ totally correct} \]

\[ \text{else } \{
\]

\[ TR = \text{triggers}(\text{nodes}(\langle \mathcal{S}, [ ] \rangle), \kappa); \]

\[ \text{if } TR = \emptyset \]

\[ \text{then FAIL} \quad \text{// permanent deadlock} \]

\[ \text{else } \{
\]

\[ \beta = \text{CHOICE}(TR); \]

\[ P = \beta; \quad \text{// build } P \]

\[ \kappa = \text{update}(\kappa, \beta); \]

\[ \text{last} = \beta; \]

\[ \text{while } \text{deadlocks}(\text{nodes}(\langle \mathcal{S}, P \rangle)) \neq \emptyset \]

\[ \text{do } \{
\]

\[ N = \text{evolutions\_after}(\langle \mathcal{S}, P \rangle, \text{last}); \]

\[ \text{if } \text{deadlocks}(N) \neq \emptyset \land \text{successes}(N) \neq \emptyset \]

\[ \text{then FAIL} \quad \text{// permanent deadlock} \]

\[ \text{else if } \text{deadlocks}(N) = \emptyset \]

\[ \text{then } \{
\]

\[ \text{undo}(\kappa, \text{last}); \]

\[ \text{last} = \text{“parent of last in } P\text{”}; \]

\[ \} \]

\[ \text{else } \{
\]

\[ TR = \text{triggers}(N, \kappa); \]

\[ \text{if } TR = \emptyset \]

\[ \text{then FAIL} \quad \text{// permanent deadlock} \]

\[ \text{else } \{
\]

\[ \beta = \text{CHOICE}(TR); \quad \text{// expand } P \]

\[ P = \text{“add } \beta \text{ to } P \text{ as a new child of } \text{last”}; \]

\[ \kappa = \text{update}(\kappa, \beta); \]

\[ \text{last} = \beta; \]

\[ \} \}

\[ \text{return}(P, \gamma); \]

\[ \} \}

---

Figure 3.8: Completion construction.
Each instance carries on the computation from the state that has been computed by all its ancestors along the branch of the tree to which it belongs. It then continues the execution from the next command after \texttt{CHOICE}.

The execution of \( x = \text{CHOICE}(S) \) spawns as many instances of the current execution as the values which are in the set \( S \). Each newly created instance inherits the state of the computation from the process executing \texttt{CHOICE}, except for the value of \( x \), for which each instance assumes one of the different values in \( S \). The set of instances forms a tree, where each instance (node) has as many children as the instances it generates.

Following Figure 3.8, and also the examples of Section 3.5.2, the algorithm can be read as follows. The input of the algorithm is the session \( S \) and the initial knowledge of the fresh names that can be used in the construction of the completion. The completion under construction \( P \) is supposed to join the session, making it closed and correct, i.e. \( \exists \gamma. \ S' = \text{join}(P, \gamma, S) \) and \( S' \) is totally correct. The mapping \( \gamma \) is easily defined by assuming that for each open variable \( x \) of \( S \), there exists a corresponding open variable \( x_P \) in \( P \), and that they are connected together by a mapping \([n/x, n/x_P]\), where \( n \) is a fresh name.

This is implemented by the function \texttt{define_mapping} \((X_S, X_P)\), which returns the mapping \( \gamma \) and the set \( \eta \) of the fresh names used to construct the mapping. The session is updated by applying the mapping, and the completion knowledge is set equal to \( \eta \).\footnote{When verification is applied to security protocols, Chapter 5, where the existence of a “particular completion” is checked, which plays the part of the intruder of the protocol, the initial \( \kappa \) can be used to provide the completion with a knowledge about the session. In particular, an intruder can be provided with some of the keys known to the participants of the protocol.}

The under construction completion \( P \) can hence be properly joined to the session by means of the mapping \( \gamma \). Consequently, in the rest of the construction, the algorithm refers to a completion whose open variables \( X_P \) have already been instantiated by \( \gamma \) and that is hence able to properly interact within the session.

If the session is deadlock free, i.e. all its traces lead to successful sessions, then the empty completion is returned. Otherwise, if no trigger is available for the deadlocked states of the traces of the session \((\text{TR} = \emptyset)\), then the session is not acceptable and the algorithm fails.

If there are triggering actions, non-deterministically one instance of the algorithm is created for each existing trigger of \( S \) \((\beta = \text{CHOICE}(\text{TR}))\). For each instance, a different tentative completion \( P \) is expanded (or initialised in the first cycle of the algorithm) with one of the possible triggering action. Each instance also updates its local copy of \( \kappa \), according to the names possibly acquired consequently to the execution of the chosen communication action \((\kappa = \text{update}(\kappa, \beta))\). The action \( \beta \), added to the completion, is also recorded as the last action taken into consideration \((\text{last} = \beta)\).

Then, each instance of the algorithm continues to explore the state space by a cycle that can either fail or terminate when the couple \( (S, P) \) results deadlock free.
The exploration of the space of the traces of \( \langle S, P \rangle \) is focused on the consequences of the last action added to \( P \) (\( N = \text{evolutions\_after}(\langle S, P \rangle, \text{last} \))).

The first condition checked is whether there are in \( N \) both success and deadlock states (\( \text{deadlocks}(N) \neq \emptyset \land \text{successes}(N) \neq \emptyset \)). This, as shown in the previous example (Figure 3.7), is a sufficient condition for the impossibility of generating a completion by expanding the so-far generated one. In fact, on the one hand the completion should be further expanded in order to eliminate deadlocks, but on the other hand, any action added to the completion will spoil the existing successful traces (for which the action that should be added is not necessary). In this case, the algorithm instance fails.

Differently, if there are not any deadlocks in \( N \), (\( \text{deadlocks}(N) = \emptyset \)), i.e. the last action used to expand \( P \) drives all the traces in which it participates to a successful state, then the completion does not need to provide the session with any other trigger after the last action executed. Hence, no action must be added to the completion as a child of \( \text{last} \). Of course this does not imply that the construction is completed, and the last but one action added to \( P \) is taken into consideration to check whether it needs to be expanded with other possible children actions. A backtracking step is performed: the knowledge is restored as it was before the execution of \( \text{last} \) (\( \text{undo}(\kappa, \text{last}) \)), and \( \text{last} \) now points to the parent of the last action added (\( \text{last} = \text{“parent of last in } P\text{”} \)). Figure 3.10 shows a case of backtracking.

The third case, the one in which there are deadlock states, but not success states after \( \text{last} \), checks for the existence of at least a trigger. If there are not triggers (\( \text{TR} = \emptyset \)) then it is not possible to expand the completion and the algorithm instance fails. Otherwise, one trigger is non-deterministically chosen (\( \beta = \text{CHOICE}(\text{TR}) \)) and used to expand \( P \). The action \( \beta \) is added as a child of the last action previously added (so that it will be executed by \( P \) after \( \text{last} \), i.e. in one of the states which belongs to \( N \), the continuation after \( \text{last} \)). Then \( \kappa \) and \( \text{last} \) are updated (\( \kappa = \text{update}(\kappa, \beta) \) and \( \text{last} = \beta \)) consequently with the chosen trigger and the cycle iterates.

The cycle iterates until the whole state space has been successfully explored (the while guard \( \text{deadlocks}(\text{nodes}(\langle S, P \rangle)) \neq \emptyset \) is false), and a completion has been constructed, or a failure has been encountered.

The non-deterministic algorithm either returns not acceptable if all its finitely many instances terminate on a FAIL point, or it returns the set of the \( P \text{s} \) computed by its instances (\( \text{return}(P, \gamma) \)). Since in order to check the acceptability of the session we do not distinguish the different completion returned, we can assume that an actual implementation of the algorithm returns one (e.g., the first computed) of the completions.

### 3.5.3 Constructing completions: examples

This section discusses some other simple examples that underline the functioning of the algorithm in different possible cases.
The session \{[\beta.\alpha + \alpha]\} of Figure 3.9 has two different completions. The session initially deadlocked offers two triggers. The nondeterministic choice of the trigger \(\overline{\beta}\) brings to the completion \([\overline{\beta}.\overline{\alpha}]\). Otherwise, in one step, the completion \([\overline{\alpha}]\) is returned.

Checking for the acceptability of the session \{[\alpha||\beta].[\bar{\alpha} + \bar{\beta}]\} of Figure 3.10 requires more steps. The session evolves in two deadlocked states, offering the triggers \(\overline{\alpha}\) and \(\overline{\beta}\). The two states generate distinct symmetric instances of the algorithm. Let us follow the one for \(\overline{\alpha}\) (the leftmost one). Having added \(\overline{\alpha}\) as last element to \(P\), five states are now reachable. The evolutions after \(\overline{\alpha}\), shaded in the figure, do not contain deadlocks, so last is backtracked until root, where the only deadlocked state offers the trigger \(\overline{\beta}\). Note, for example, that in the case in which the completion does not have access to the channel name of \(\beta\), no trigger exists, and hence the
3.5. VERIFYING ACCEPTABILITY

Figure 3.10: Algorithm $A$ at work - III.

session would result in not being acceptable. After having added $\beta$ to $P$ as a child of $root$, the evolutions after $\beta$, first, and then all the reachable states are checked in two backtracking steps, and they do not present deadlocks. The completion $[\bar{\alpha} + \beta]$ is returned.

In Figure 3.11 it is possible to see how local choices force the global interaction. The session $\{[\tau.\alpha + \tau.\beta]\}$ can “autonomously decide” to evolve in two deadlocked sessions. Each of them offers a trigger, producing two different nondeterministic instances of the algorithm. The leftmost of them chooses $\alpha$ as a trigger to expand the completion. The continuations after $\alpha$ consist of one successful state. Hence last backtracks until assuming the value root. Now, a deadlocked state with trigger $\beta$ can be reached. Then, $\beta$ is added as a child of root, and, since there are not deadlocks the completion $[\alpha + \beta]$ is returned in the left bottom step of this instance. The other instance returns the structurally equivalent completion $[\beta + \alpha]$. 
3.5.4 Complexity

Given a session \( S \), the algorithm \( \mathcal{A} \) exhaustively explores the states which can be reached by the traces of \( S \). The computational complexity of the algorithm can hence be measured in terms of the number of the states which can be reached by the traces of \( S \). This is clearly a defective esteem, both because each non-deterministic instance of the algorithm can explore a vast majority of the states, and because the incremental construction of the completion adds new states by removing deadlocks.

Nonetheless, this esteem, even if defective, is sufficient to state that the algorithm \( \mathcal{A} \) belongs to the \( \text{EXP} \) class of complexity, i.e. the class of problems that can be solved in a time that is, in the worst case, exponential with respect to the dimension of the input instance of the problem.

A natural measure of the dimension of the input instance, i.e. of the session \( S \), is the structural complexity of the interaction patterns that are present in the session.
3.5. VERIFYING ACCEPTABILITY

Structural complexity of an interaction pattern can be roughly given by the number of the actions the pattern contains. The combinatorial way in which actions can be combined gives rise to an exponential growth of the number of the states that the session can reach.

For instance, the session in Figure 3.10 clearly illustrates the phenomenon. It contains the two simple interaction patterns \( )[\alpha||\beta] \) and \( )[\bar{\alpha} + \bar{\beta}] \), having hence a dimension of 4. The two successful instances of \( \mathcal{A} \) visit a total of 33 states. It must anyway be remembered that in the general case the problem of verifying properties about the semantics of a concurrent calculus is inherently exponential.

Implementing the algorithm, it is possible to introduce some optimisations. In particular, it would be worth implementing a strategy (or an heuristic) for expanding the tree of the non-deterministic instances so as to rapidly converge towards one of the existing solutions. For example, back to Figure 3.10, the two instances build two symmetric (structurally equivalent) completions by performing the same choices in a different order.

Other than depending on an intelligent implementation, practical applicability of the algorithm depends on its correct usage inside the proposed methodology. Acceptability of a system can be checked either during system design and maintenance or in a more dynamic scenario.

Checking a system during the design phase is an off-line static procedure. This is a standard case of model checking, where a system is analysed by verifying the existence of a model of the system that enforces the desired properties. The field of finite state model checking has been largely investigated in the literature, and the last decade advances in techniques and implementations have made the verification of commercial-size applications possible, whose models have a huge number of reachable states, see [CW+96] for a survey, and, more in general, Chapter 2.

Dynamically verifying a running system, for example to check the compatibility of an updated component within a system, or to grant the access of a mobile component to an open site, is a process that may have tighter efficiency constraints. In this respect, partitioning the behaviour of a component into interaction patterns of a suitable granularity, i.e. the complexity of the behaviour that an interaction pattern represents, may be a solution when efficiency constraints are an issue. Finite, properly dimensioned interaction patterns do not allow for an unbound growth of the number of the reachable states. The dimension of the patterns can be used to control computational efficiency.

Moreover, it must be remembered that the finiteness of the proposed approach is motivated, not only by efficiency issues, but also by the inherent incompleteness of open systems, as a way to limit the analysis to what is currently observable in the system.

A further enhancement is represented by symbolic model checking, that has been applied in Chapter 5 to the verification of properties of security protocols, as a means to limit potential unboundedness of the traces when dealing with infinite sets of data exchanged.
3.5.5 Correctness of the algorithm

The algorithm $\mathcal{A}$ can be proved correct according to the following property:

“the algorithm always terminates and it returns a completion if and only if the session is acceptable.”

The property is proved in three steps: 7:

**Theorem 3.4** (Correctness of the algorithm) For each session $S$:

1. $\mathcal{A}(S, \kappa)$ terminates.
2. If $P \in \mathcal{A}(S, \kappa)$ then $S$ is acceptable.
3. If $S$ is acceptable then $\mathcal{A}(S, \kappa) \neq \text{not acceptable}$.

**proof of 1** Termination of the non-deterministic algorithm is proved by showing that:

- **The number of instances of $\mathcal{A}(S, \kappa)$ is finite.** Each new instance is spawned correspondingly to a chosen trigger that is added to the under construction completion $P$. According to the definition of *triggers*, the same action in $S$ can not be used twice for generating a trigger. In fact, after the trigger has been added to the under construction completion $P$, it will never be removed from $P$ (and hence the corresponding generating action does not belong to a deadlock state anymore). It follows that any given instance can have at most a finite number of descendant (each one corresponding to one of the finite possible triggers in the session), and hence the tree of the instances of the algorithm is finite.

- **Each instance of $\mathcal{A}(S, \kappa)$ terminates.** Termination of each instance of the algorithm reduces to termination of the *while* cycle. At each iteration of the *while* cycle, every instance can:
  i) terminate with failure, or
  ii) modify the value of *last*, or
  iii) non-deterministically choose new triggers $\beta$s and generate the corresponding non-deterministic instances (correspondingly the derivation trees for $S \cup [P']$, where $P'$ has been expanded with a $\beta$, is expanded).

---

7As explained in Section 3.5.2, the outcome of the non-deterministic algorithm can be either a failure message, or a set of computed completions. With a slight abuse of notation, we indicate with $\mathcal{A}(S, \kappa) = \text{not acceptable}$ the first case (and $\mathcal{A}(S, \kappa) \neq \text{not acceptable}$ stands for “$\mathcal{A}(S, \kappa) = \text{not acceptable}$ does not occur”), while with $P \in \mathcal{A}(S, \kappa)$ we indicate that $P$ belongs to the set of non-deterministically computed completions.
3.5. VERIFYING ACCEPTABILITY

Case i) directly leads to termination.

Case ii) can be repeated only a finite number of consecutive times before case i) or iii) occurs again. At each consecutive iteration due to case ii) the value of last assumes the value of the action that is the ancestor of last in $P$, until it will eventually assume the value of root, but this can not happen two consecutive times. In fact, let us suppose case ii) occurs two consecutive times with last = root. The while cycle is entered with last = root. Then, because of the while guard, a deadlocked state $⟨R\text{lock}, P\text{lock}⟩$ must exists, such that $⟨S, P⟩ \mapsto^∗ ⟨R\text{lock}, P\text{lock}⟩$. Suppose, in order to have again last = “parent of last” = root, that $\text{deadlocks(}evolutions_{\text{after}}(⟨S, P⟩, root)\text{)} = \emptyset$, where $evolutions_{\text{after}}(⟨S, P⟩, root)) = \{⟨R, P⟩ | ⟨S, P⟩ \mapsto^∗ ⟨R, P⟩ \land N' = (β, β)^*\}$ (i.e., the traces done without interacting with $P$). Then, a deadlock must have been introduced by the interaction with $P$, i.e. a derivation $⟨S, P⟩ \mapsto^∗ ⟨S', P'⟩ \mapsto^* ⟨R', P'⟩ \mapsto^* ⟨R\text{lock}, P\text{lock}⟩$ must exists, for an action $α$ previously added to $P$. But this is not compatible with the fact that the cycle is entered with last = root, since after the selection of the trigger $α$, it was holding $⟨R\text{lock}, P\text{lock}⟩ \in \text{deadlocks(}evolutions_{\text{after}}(⟨S, P⟩, α)\text{)} \neq \emptyset$, preventing last from ever assuming the value root. Finally, note that the number of possible triggers in the session is not influenced by the iterations due to case ii).

Case iii) eventually leads to termination. As observed above, this case, i.e. the spawning of new instances of the algorithm, decreases the number of total instances that can be spawned as descendant of the current instance. It means that at each time the while cycle iterates because of case iii), the number of possible trigger decreases, until eventually reaching the value of 0. In that case the while can not iterate anymore because of case iii), and if the conditions for case iii) still apply (i.e. there are deadlocks but not triggers) the iteration fails.

Summing up, within a finite number of consecutive iteration, the while cycle necessarily decreases the number of possible triggers in the session, which are initially finite. When the number becomes 0, the cycle must exit after a finite number of iterations (due to case ii)).

\[\square\]

**proof of 2** Straightforward, by the boolean condition of the while cycle, since the definition of acceptability reads as $\text{deadlocks(}nodes(⟨S, P⟩)\text{)} = \emptyset$. 

\[\square\]
The proof of point 3 is a bit more complex, and it is based on the existence of an instance of the algorithm which constantly “approximates” a standard form completion $P$, whose existence is granted by the acceptability of the session. A failure of the instance (viz., of all its descendant) would imply the non existence of $P$.

Accordingly to the next definition, an interaction pattern $P$ (i.e. the existing normal form completion) extends $Q$ (i.e. the under construction completion) with respect to a session $S$ if the traces of $⟨S, Q⟩$ can be “mapped” in those of $⟨S, P⟩$.

**Definition 3.13 ($≪_S$)** Given two interaction patterns $Q$ and $P$ and a session $S$, then $Q ≪_S P$ ($P$ extends $Q$ with respect to $S$) if and only if

\[
\forall S', Q', Λ_1 : ⟨S, Q⟩ \xrightarrow{Λ_1}^* ⟨S', Q'⟩
\]

\[
\Downarrow
\]

\[
\exists P', Λ_2 : ⟨S, P⟩ \xrightarrow{Λ_2}^* ⟨S', P'⟩ \land \bar{Λ}_1 = \bar{Λ}_2,
\]

where $\bar{Λ}_1$ (resp. $\bar{Λ}_2$) indicates the (sub-)sequence of actions done by $Q$ (resp. $P$) in the trace relative to $Λ_1$ (resp. $Λ_2$).

The proof is divided in two complementary fact, respectively proved by the following two lemmas: under the hypothesis that a standard form completion $P$ exists then

- if at the beginning of the while cycle the partial completion constructed so far, $P_{i-1}$, satisfies $P_{i-1} ≪_S P$, the cycle is iterated, and the cycle does not fail, then the partial completion $P_i$, produced at the end of the cycle, satisfies $P_i ≪_S P$, and

- if at the beginning of the while cycle the partial completion constructed so far, $P_{i-1}$, satisfies $P_{i-1} ≪_S P$, and the cycle is iterated (i.e. there are still deadlocks) then the iteration can not fail.

In the following proofs we will uniformly indicate with [] either a successful session of a pattern $P \equiv ()[0]$.

**Lemma 3.2** Let $S$ be a session, $P$ a standard form completion and $P_{i-1}$ the value of the completion so far constructed at the beginning of the $i$-th iteration of the while cycle. If the cycle is iterated (the while guard is satisfied) and the iteration does not fail, then, for at least an instance of the algorithm (with $P_i$ the completion produced at the end of the cycle):

\[
P_{i-1} ≪_S P \Rightarrow P_i ≪_S P
\]
By hypothesis, $P_{i-1} \ll_S P$ and the cycle is executed for the $i$-th time without failing. At the end of the $i-th$ iteration, $P_i$ can have been defined in two ways:

- $P_i = P_{i-1}$, in case of backtrack, and, trivially, $P_i \ll_S P$, or

- $\exists \langle S', P'_{i-1} \rangle \in \text{deadlocks}(N) \neq \emptyset$ (since the $i$-th cycle has been executed), such that

$$\langle S, P_{i-1} \rangle \xrightarrow{\Lambda_{1,\text{last}}} * \langle S', P'_{i-1} \rangle \not\rightarrow$$

and,

$$\langle S, P \rangle \xrightarrow{\Lambda_{2,\text{last}}} * \langle S', P' \rangle \xrightarrow{\alpha} \langle S'', P'' \rangle \xrightarrow{\ast} \langle \emptyset \rangle$$

by hypothesis

$P$ is a completion

with $\bar{\Lambda}_1 = \bar{\Lambda}_2$. It follows that $\exists \alpha \in \text{triggers}(\langle S', P'_{i-1} \rangle \in N) \land P' \xrightarrow{\alpha} P''$, in fact $S'$ can “move” together with $P$. It follows that, for an instance, $P_i = "\text{add } \alpha \text{ as child of last (in } P_{i-1})"$.

Let us now show that $P_i \ll_S P$, by considering a state reachable from $\langle S, P_i \rangle$:

$$\langle S, P_i \rangle \xrightarrow{\Lambda_1} \langle S, P_i' \rangle.$$

Two cases are possible:

- $(-\alpha) \notin \Lambda_1$, i.e., the added action $\alpha$ does not occur in the trace. In this case, it holds

$$\langle S, P_{i-1} \rangle \xrightarrow{\alpha \in \Lambda_1} \langle S, P'_{i-1} \rangle,$$

and, by hypothesis,

$$\langle S, P \rangle \xrightarrow{\Lambda_2} \langle S, P' \rangle \land \bar{\Lambda}_1 = \bar{\Lambda}_2,$$

and hence $P_i \ll_S P$.

- $(-\alpha) \in \Lambda$. By construction, $\alpha$ is the last action done by $P_i$ in $\bar{\Lambda}_i$, and, hence $P_i$ coincides with $P_{i-1}$ along $\Lambda_i$, except for the last action $\alpha$. Then
\[ \langle S, P \rangle \xrightarrow{\Lambda_{1, \text{last}}} \ast \langle S', P' \rangle \xrightarrow{\alpha} \langle S, P'' \rangle \]

\[ \Downarrow \]

\[ \langle S, P \rangle \xrightarrow{\Lambda_{1, \text{last}}} \ast \langle S', P' \rangle \]

\[ \Downarrow \]

\[ \langle S, P \rangle \xrightarrow{\Lambda'_{\text{last}'}} \ast \langle S', P' \rangle \xrightarrow{\alpha} \langle \tilde{S}, P'' \rangle \]

and hence \( P_i \ll_s P \).

Then, if the \( i \)-th cycle is executed, from \( P_{i-1} \ll_s P \) it follows

\[ P_i \ll_s P \]

\[ \square \]

**Lemma 3.3** Let \( S \) be a (not totally correct) session, \( P \) a standard form completion and \( P_{i-1} \neq [] \) the value of the so far constructed completion at the beginning of the \( i \)-th iteration of the while cycle. Then

\[ \text{deadlocks}(\text{nodes}(\langle S, P_{i-1} \rangle)) \neq \emptyset \land P_{i-1} \ll_s P \]

\[ \Downarrow \]

"the \( i \)-th iteration of the while cycle does not fail"

**Proof.**

Within the cycle, failure can occur under two conditions. We show that a failure in any of the two cases contrasts with the hypothesis of \( P \) being a completion.

1. \( \text{deadlocks}(N) \neq \emptyset \land \text{success}(N) \neq \emptyset \)

In this case, there exist \( \Lambda_s \) and \( \Lambda_d \) such that:

\[ \langle S, P_{i-1} \rangle \xrightarrow{\Lambda_s} \langle [], [] \rangle \text{ and } \langle S, P_{i-1} \rangle \xrightarrow{\Lambda_d} \langle S', P'_{i-1} \rangle \neq . \]

Both \( \langle [], [] \rangle \) and \( \langle S', P'_{i-1} \rangle \) belong to \( N = \text{evolutions after}(\langle S, P \rangle, \text{last}) \) the set of states reached by a trace in which \( \text{last} \) is the last action done by the completion. Depending on the value of \( \text{last} \), two cases must be further considered:
• last \neq \text{root}

In this case \( \Lambda_s = \lambda^* \). \((-\text{last})\). \(\{ (\alpha, \bar{\alpha}) \}^*\), i.e. \( \Lambda_s \) consists of a number of generic actions, followed by \text{last}, followed exclusively by actions that \( \mathcal{S} \) can autonomously perform.

As well as in \( \Lambda_d \), \text{last} is the last action that \( P_{i-1} \) can do, then \( P'_{i-1} = \emptyset \), while \( \mathcal{S}' \neq \emptyset \) is a deadlock state.

From \( P_{i-1} \ll P \), it follows that \( \exists \Lambda'_d: \langle \mathcal{S}, P \rangle \overset{\mathcal{N}_d}{\rightarrow} \langle \emptyset, \emptyset \rangle \), and \( \Lambda_d = \Lambda'_d \), i.e. \text{last} is an action done by \( P \) through the trace, and for what said before, \text{last} is the last action of \( P \) along \( \Lambda'_d \), i.e. \( P' \) can not perform any action in \( \langle \mathcal{S}', P' \rangle \). Then, \( \langle \mathcal{S}' \neq \emptyset, P' \rangle \not\twoheadrightarrow \), i.e. there is a deadlock trace of \( \langle \mathcal{S}, P \rangle \) not leading to a successful session, against the assumption of \( P \) being a completion.

• last = \text{root}

By definition of \( \mathcal{N} = \text{evolutions}_\text{after}(\langle \mathcal{S}, P_{i-1} \rangle, \text{root}) \), only the states \( \langle \mathcal{S}', P_{i-1} \rangle \), such that \( S \mapsto^* S' \), are in \( \mathcal{N} \). Since, by hypothesis, \( P_{i-1} \neq \emptyset \), it follows that \( \text{success}(\mathcal{N}) = \emptyset \), and hence the failure condition \( \text{deadlocks}(\mathcal{N}) \neq \emptyset \land \text{success}(\mathcal{N}) \neq \emptyset \) does not hold when last = root.

Under the hypothesis of the lemma, then, the condition \( \text{deadlocks}(\mathcal{N}) \neq \emptyset \land \text{success}(\mathcal{N}) \neq \emptyset \) can not cause a failure.

2. \text{deadlocks}(\mathcal{N}) \neq \emptyset \land \text{success}(\mathcal{N}) = \emptyset \land \text{triggers}(\mathcal{N}, \kappa) = \emptyset

Since the condition \( \text{triggers}(\mathcal{N}, \kappa) = \emptyset \) depends on the set \( \mathcal{N} \), which depends on the value of last, again two cases need to be considered.

• last = \text{root}

In this case,

\( \exists \Lambda, \langle \mathcal{S}', P'_{n-1} \rangle \in \text{deadlocks}(\mathcal{N}) : \langle \mathcal{S}, P_{i-1} \rangle \overset{\Lambda}{\rightarrow} \langle \mathcal{S}', P'_{i-1} \rangle \).

The definition of \( \mathcal{N} = \text{evolutions}_\text{after}(\langle \mathcal{S}, P_{i-1} \rangle, \text{root}) \) implies that \( P_{i-1} \) does not perform any action in the trace, and hence \( S \mapsto^* S' \). We can assume, without loss of generality, that \( \mathcal{S}' \neq \emptyset \), in fact it must exist a trace in which \( \mathcal{S}' \) is not successful (otherwise \( \mathcal{S} \) is totally correct and the while cycle does not start).
Note that the condition \(\text{triggers}(N, \kappa) = \emptyset\) implies that
\[
\neg (\exists Q \in S'. \ Q \xrightarrow{\alpha} Q' \land ch(\alpha) \in \kappa),
\]
i.e. either \(\not\exists Q \in S'. \ Q \xrightarrow{\alpha} Q'\) or \(ch(\alpha) \not\in \kappa\).

Since by hypothesis \(P_i \ll P\),
\[
\exists \Lambda': \langle S, P \rangle \xrightarrow{\Lambda'} \langle S', P' \rangle,
\]
and, being \(P\) a completion, \(\exists Q \in S'. \ Q \xrightarrow{\alpha} Q'\) such that \(\langle S', P' \rangle \xrightarrow{\alpha} \langle S'', P'' \rangle \xrightarrow{*} \langle [], [] \rangle\).

It then must be \(ch(\alpha) \not\in \kappa\), while \(P'\) “knows” \(ch(\alpha)\). Since, \(P_{i-1}\) initially knows all the (open) channel that \(P\) may know (via the initial knowledge \(\kappa\)), and \(P_{i-1}\) and \(P\) have performed the same communication actions \(\bar{\Lambda} = \bar{\Lambda'}\), it can not happen, against what deduced, that \(P\) “knows” \(ch(\alpha)\), while \(ch(\alpha) \not\in \kappa\).

- last = \(\alpha\)

In this case,
\[
\exists \Lambda, \langle S', P'_{n-1} \rangle \in \text{deadlocks}(N) : \langle S, P_{i-1} \rangle \xrightarrow{\Lambda} \langle S', [] \rangle,
\]
since last is the last action done by \(P_{i-1}\) (which, by construction, can not perform any other action after last). Then \(S' \neq []\).

Since by hypothesis \(P_i \ll P\),
\[
\exists \Lambda': \langle S, P \rangle \xrightarrow{\Lambda'} \langle S', P' \rangle,
\]
and, being \(P\) a completion, \(\exists Q \in S'. \ Q \xrightarrow{\alpha} Q'\) such that \(\langle S', P' \rangle \xrightarrow{\alpha} \langle S'', P'' \rangle \xrightarrow{*} \langle [], [] \rangle\).

According to the same considerations about the knowledge of \(P_{i-1}\) with respect to that of \(P\) done in the previous point, it follows, against the hypothesis, that \(\bar{\alpha} \in \text{triggers}(N, \kappa)\).

Under the hypothesis of the lemma, then, the condition \(\text{deadlocks}(N) \neq \emptyset \land \text{success}(N) = \emptyset \land \text{triggers}(N, \kappa) = \emptyset\) can not cause a failure.

\[\Box\]

Finally, the proof of point 3 shows that if a completion exists, than there is an instance that can not fail, and, since it terminates, it must terminate with the successful construction of a completion.
3.5. VERIFYING ACCEPTABILITY

**proof of 3** Let $P_{i-1}$ be a completion partially constructed before entering the $i$-th iteration of the cycle, and $P$ the existing (standard form) completion.

The algorithm can either fail before the while cycle or inside it. In the first case it is trivial to observe that $S$ can not be acceptable, against the hypothesis.

Otherwise, among the possible triggers of the not totally correct session $S$, there exists an action $\alpha$ such that $P \xrightarrow{\alpha} P'$ (otherwise $P$ could not trigger the deadlock state). For the instance that chooses $\alpha$ as a trigger for starting the construction of $P_0$, it obviously holds $P_0 \llc_S P$, before entering the cycle.

Since, each iteration of the while cycle starting with $P_{i-1} \llc_S P$,

- by Lemma 3.3, does not fail, and,
- by Lemma 3.2, can be computed by an instance yielding $P_i$ such that $P_i \llc_S P$,

then, observing that at the first iteration $P_0 \llc_S P$, we can conclude that there is a sequence of instances through which the while cycle does not fail, and, since it terminates, by point 1 of correctness, it must exit because, at a given iteration $i$, the while condition $\text{deadlocks}(\text{nodes}(\langle S, P_i \rangle)) \neq \emptyset$ is false. It follows that corresponding instance returns $P_i$, that is a completion.

\[\square\]
Chapter 4
Adaptors

Contents

4.1 Motivations and overview ......................................................... 95
4.2 Component interfaces .............................................................. 97
4.3 Adaptor specification ............................................................... 99
4.4 Adaptor derivation ................................................................. 104
4.5 An example of adaptation ........................................................ 110
4.6 About the role of the adaptor ................................................... 113
4.7 Concluding remarks ............................................................... 116

Chapter 3 introduced a framework for composing components into open systems (sessions) and reasoning about associated correctness properties. It has been shown how acceptability can be used to enforce the desired “well-behaving” of an open system, by preventing the access to those components that may spoil the “balance” of a system by introducing unrecoverable errors (rule (in), Figure 3.9).

Even if acceptability is a desirable invariant in the life of a system, it may sometimes result too strict. Indeed, with the aim of preserving the correctness of the system, it may also exclude components that “could” otherwise correctly behave under certain hypothesis.

In this chapter the way in which the proposed framework can be extended so as to let interoperate components that would not spoil the correctness of a session if properly adapted each other is investigated. By exploiting a variant of the algorithm $\mathcal{A}$ of Section 3.5.2, it will be shown how to build a special completion, called adaptor, that adapts components with mismatching behaviours in a properly working system.

The construction of an adaptor starts from the mismatching components and a partial, high-level specification of the required mediation to let them properly interoperate. This specification is called behaviour mapping, or more simply mapping.
An ad-hoc high-level notation has been introduced to express the mapping among the different languages spoken by the components.

Mapping design requires, in the general case, intelligent (human) supervision. In this dissertation, we do not address mapping design as such, but rather we provide an abstract notation intended to facilitate mapping specification by human designers. More precisely, the notation permits the burden of actual adaptor construction to be abstracted away from the process of mapping specification.

Given the mismatching components and the mapping, an adaptor is a component such that, when connected with the components, produces a properly functioning system, in which interaction is carried on according to the mapping.

Once that the required adaptation has been specified in terms of a mapping, if an adaptor exists, then it can be generated by an automatic procedure, in a similar way as completions are generated. Indeed the adaptor is a component which makes the open system consisting of the mismatching components correct.

Moreover, the algorithmic construction, i.e. a variant of the completion construction algorithm (Section 3.5.2), guarantees the correspondence of the adaptor with its specification, i.e. the mapping.

In some cases, it may be worth enhancing the expressiveness of the mapping language, reducing it abstractness, in order to obtain a finer control of the adaptor usage of data and, hence, further constrain the construction procedure. This is done by adding simple properties to the mapping language. The modifications consequently required by the algorithm are in general minimal and modular, so that, in principle, ad-hoc properties can be easily defined by the mapping specifier.

The rest of the chapter is organised as follows. In Section 4.1, adaptors are motivated within the current practice and research, in particular in the context of Component Based Software Engineering. Section 4.2 introduces the syntax of a standard Interface Description Language extended with a IP-calculus based behavioural description. Mappings as adaptor specifications are presented in Section 4.3, while automatic adaptor generation is explained in Section 4.4. The applicability of the whole methodology is illustrated in Section 4.5, where a realistic case of adaptation between two components employing different file transmission protocols is analysed. Section 4.6 shows how to use simple properties about data in order to further specify adaptor construction. The final Section 4.7 is devoted to discussing related work and to making some concluding remarks.

The first results of this work appeared in [BBC02b] and were extended in [BBC02a], properties for a finer specification of the adaptor have been studied in [BBC02c], while the whole methodology is illustrated in [BBC03].
4.1 Motivations and overview

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [Cam99, Hei99, GS01]. The possibility for application builders to easily adapt off-the-shelf software components to work properly within their application is a must for the creation of a true component marketplace and for component deployment in general [BW98].

Available component-oriented platforms (e.g., CORBA [OMGa], COM [Cha96], JavaBeans [SUNa]) address software interoperability by using Interface Description Languages (IDLs). The provision of an IDL interface defining the signature of the methods offered (and possibly required) by a component is an important step towards software integration. IDL interfaces highlight signature mismatches between components in the perspective of adapting or wrapping them to overcome such differences. Other than problems that may arise at the signature level, like for example method names and parameter usage discordances, mismatches may also occur at the protocol level, because of the ordering of exchanged messages and of blocking conditions [VHT00], that is, because of differences in component behaviours.

As we have already pointed out, available component platforms do not provide suitable means for describing the interaction behaviour of a component. Consequently, behaviour mismatches can only be manually adapted.

While case-based testing can be performed to check the compatibility of the behaviour of components, more rigorous techniques are needed to lift component integration from hand-crafting to an engineering activity, based on the rigorous verification of systems consisting of large numbers of components that dynamically interact with one another [CGL94]. This need is even more stringent when assembling third-party components, as happens in CBSE.

The problem of component adaptation, in the process of designing and formally verifying applications, has been the subject of intensive attention in the last few years. A number of practice-oriented studies have been devoted to analyse different issues to be faced when adapting a third-party component for a (possibly radically) different use (e.g., see [GAO95, DR97, KWS01]). A formal foundation for component adaptation was set by Yellin and Strom in their seminal paper [YS97]. They introduced the notion of adaptor as a software entity capable of letting two components with mismatching behaviours interoperate, using finite state machines (FSM) for specifying component behaviour.

The approach of extending IDLs with FMS-based behaviour description has been followed by several other authors, like, e.g., the recent [Reu01], who advocates the tractability of FMS as a strong motivation to prefer them to other formalisms.

In this context, IP-calculus has been applied to support a formal methodology for adapting components with possibly mismatching interaction behaviours. The interest of this experiment is twofold:

1. to validate the expressiveness of IP-calculus, studying how it can be used to
deal with a problem that is both an industrial requirement, and also an open research problem, and

2. to contribute to the ongoing research on adaptation by proposing both the abstract mapping specification language, and a more expressive calculus than FSM. For example, such a calculus permits the reconfiguration of the communication network to be modeled, while it enjoys a similar effectiveness to that of FSM.

In order to present a setting closer to the traditional “object-like” context of CBSE, a more object-oriented syntax for IP-calculus has been adopted. Very simply, the output communication \( \text{out}(\text{ref}, \text{data}) \) becomes the reference call (to a method but also, as usual, to a channel) \( \text{ref}!(\text{data}) \), while the input action \( \text{in}(\text{ref}, \text{data}) \) represents a method that, at a given instant, the component offers to its environment: \( \text{ref}?(\text{data}) \) (or a channel on which it is ready to receive data). In a word, references can be thought as method names. In this context, for the sake of simplicity, variables are not upper-case whenever the usage clearly distinguishes them (typically, coherently with the “method call” reading, each input action instantiates all its (variable) parameters).

Component interfaces consist of a static signature, describing the functionalities offered and required by the component, and a behaviour specification describing the interaction protocol followed by the component. The couple static interface and protocol, expressed by an interaction pattern, is called role. The interface of a component may consist of more roles, each one devoted to a specific facet of the behaviour of the component. Roles can be, informally, seen as spatial partitions of the behaviour of a component, likewise interaction patterns can be seen as temporal partitions. Analogously to [CFTV99], in order to simplify adaptor construction, adaptation works role-wise, in the sense that roles can only be adapted in pairs, by means of the mediation of an “in-between” adaptor for each couple of corresponding roles of different components.

Adaptation may require both syntactical and semantical adjustments, and adaptors may need not only to behave as translators, but also to actively facilitate the interoperation of two roles. For example, beyond reordering and name translations, an adaptor may also record data and, when required, provide them, or forget those that will not be required by the components. Moreover, an adaptor may also be used to enforce the desirable property of encapsulation, by letting the components know as little as possible about each other. For example, the adaptor may hide the names of the methods of a component in order to avoid the other component directly accessing them, possibly following an uncontrolled protocol.

A distinguishing feature of the mapping representation is that it is abstract, hiding from the mapping designer a lot of procedural details that will be automatically resolved by the algorithmic adaptor construction.

Given the mapping and the interfaces of the components, a concrete adaptor is automatically generated by exhaustively trying to build a component (i.e. a
4.2. COMPONENT INTERFACES

Completion) which will allow the initial components to interoperate, while satisfying the given mapping. Constraints due to mapping specification may lead the process to failure.

The advantage of separating adaptor specification and derivation is to automate the error-prone, time-consuming task of generating a detailed implementation of a correct adaptor, while simplifying the task of the (human) software developer. The automatic adaptor construction proceeds along the line of the algorithm for checking acceptability of Section 3.5.2. Basically, the adaptor is a particular completion for the interaction patterns of the two components, which, while it enforces the correctness of the composition, translates the different languages spoken by the two.

4.2 Component interfaces

Component interfaces are described as a set of roles, each one devoted to a specific facet of the behaviour of the component. A role consists of

- a description of the component at the signature level, similar to traditional IDL descriptions. The signature interface of a component role declares a set of input and output actions the component offers to its environment. These actions can be seen as the set of messages sent and received by the role (representing the methods that the component offers and invokes, the values or exceptions returned, etc.). Notice that the IDLs of most of the current component architectures represent only the services that the component offers to its environment (that is, the set of its output actions), while we explicitly represent also the services required by the component, as a set of input actions.

Both input and output actions may have parameters, representing the data interchanged in the communication. Parameters can be typed in order to allow for type-checking. For our purposes of describing component adaptation, it is sufficient to distinguish between just two types: Link—representing channels through which messages can be sent and received—and Data representing any other data value. As usual, as shown in Chapter 5, the extension to more complex type systems is straightforward.

- a description of the component at the protocol level, in terms of interaction patterns of the IP-calculus (Chapter 3).

Roles can have a name, and encompass the two descriptions, according to the following syntax:

```
role roleName = {
  signature input and output actions
  behaviour interaction pattern
}
```
Intuitively speaking, like interaction patterns can be seen as projections “on
time” of the behaviour of a component, roles can be seen as projections “on space”,
possibly representing functional units of the component. For example, a data-
repository component may have a role for interfacing itself with a terminal, one
for interacting with a query engine, another one for requesting the printing of doc-
uments.

In modeling adaptation, open variables and the join operation loose most of
their significance. We adopt the simplifying, reasonable, hypothesis that an adaptor
has access to all the references of the components to be adapted. For this reason,
open variables will not occur in the interaction patterns of the roles. Clearly, this
presupposes that mappings are “correct”, for example in the sense that they do not
specify access to private references.

As a first example of decomposition of a component interface in more roles,
let us consider the case of a component ($\texttt{Reader}$) which sequentially reads a file.
File items are received with an action $\texttt{read?}(x)$ — the end-of-file condition being
represented by a special value $\texttt{EOF}$. Suppose that the component may decide to break
the transmission at any time by sending an action $\texttt{break!}()$. After the file item has
been read, the component copies it to disk, using actions $\texttt{fwrite!}$ and $\texttt{fclose!}$.

This behaviour in recursive full $\pi$-calculus is:

$$
\texttt{Reader} = \texttt{read?}(x). ([x!=\texttt{EOF}] \texttt{fwrite!}(x). \texttt{Reader} + [x=\texttt{EOF}] \texttt{fclose!}(). 0)
+ \tau. \texttt{break!}(). \texttt{fclose!}(). 0
$$

Now, instead of writing a single (but in fact, more complex) pattern for representing
the component, we will partition its behaviour into roles, one for describing how it
reads the file and the other describing its interaction with the file system, whose
interaction patterns, respectively, are:

$$
\texttt{read?}(x). 0 + \tau. \texttt{break!}(). 0 \quad \text{// R1}
$$

$$
\tau. \texttt{fwrite!}(\texttt{data}). 0 + \tau. \texttt{fclose!}(). 0 \quad \text{// R2}
$$

The two patterns will belong to different roles which will be connected, possibly
via an adaptor, to the corresponding roles of a disk manager component and a file
manager component, respectively. Thus, we allow for a modular representation and
analysis of behaviour, each role representing the reader in its interaction with a
different component. Hence, while $\texttt{Reader}$ chooses to send either a $\texttt{fwrite!}$ or a
$\texttt{fclose!}$ on the basis of the reception of data or end-of-file, the interaction pattern
of the role R2, i.e. the behaviour of the reader with respect to the file manager,
hides this choice within an internal action (local choice).

The special characteristics of mobility which are present in the $\pi$-calculus allow
the creation and transmission of link names which can be used later for communi-
cation. This determines that the signature interface of an interaction pattern is not
fixed (like in other process algebras or in object-oriented environments), but it can be extended because of link-passing.

As an example of these extensible interfaces, consider the pattern below, which specifies the behaviour of a component accepting queries in which a specific channel for returning the requested information is indicated. The component can also raise an exception due to internal reasons (here represented by a tau action), according to the following interaction pattern:

\[
\text{query?(return). return!(info). 0 + tau. exception!(). 0}
\]

Note that in the above pattern, action return!(info) is not immediately available, since it is “enabled” by the preceding input action. This dynamic is described in the signature interface of a component by means of the operator ‘>’ (read as “before”), which explicitly represents dependencies between link names in the interface. For the case under consideration, the interface of the component, showing the dependency between query?(Link return) and return!(Data info) in its signature interface, will be written as:

\[
\text{role QueryServer = }
\text{signature}
\quad \text{query?(Link return) > return!(Data info);}
\quad \text{exception!();}
\text{behaviour}
\quad \text{query?(return). return!(info). 0 + tau. exception!(). 0 }
\]

The presence of the > operator is intended to facilitate the definition of mappings from the static interfaces only. In general it is not possible to design a proper mapping without taking into consideration the real protocol followed by the components. At any rate, it may sometimes be worth, however, trying to determine a mapping by looking at the static interfaces only, for example in order to rapidly develop a prototype or in those cases in which a behaviour description is not available, either because of a legacy component or because of encapsulation and secrecy reasons. Even if the algorithmic procedure is anyway in charge of guaranteeing the correctness of the adaptor, knowing the dependencies among the references of the interface may facilitate the design of a significant mapping.

### 4.3 Adaptor specification

Adaptation, in its generality, is a hard problem which involves a large amount of domain knowledge and may require complex reasoning. In this context, our approach aims to provide a methodology for specifying the required adaptation between two components in a general and abstract way. In this section we will illustrate a simple and abstract language which permits us to describe the intended mapping among the functionalities of (the two roles of) the components to be adapted.
We first observe that adaptation does not simply amount to substituting reference names. Consider for instance a component \( P \) that requests a file by means of an \url, and a repository \( Q \) that first receives the \url and then returns the corresponding file. Their interaction patterns are, respectively:

\[
\begin{align*}
\text{request!}(\url) & . \text{reply?}(\text{page}). 0 \quad \text{// } P \\
\text{query?}(\text{address}) & . \text{return!}(\text{file}). 0 \quad \text{// } Q
\end{align*}
\]

The connection between \texttt{request!} and \texttt{query?}, and between \texttt{reply?} and \texttt{return!} could be defined by the substitution:

\[
\sigma = [\text{request} \rightarrow t_1, \text{query} \rightarrow t_1, \text{reply} \rightarrow t_2, \text{return} \rightarrow t_2]
\]

that allows their interoperation. Notice that, after applying the substitution, the communication between \( P_\sigma \) and \( Q_\sigma \) would be direct and unfiltered, since they will share reference names. However this contrasts with encapsulation principles as, in general, one would like neither to modify the components nor to allow the sharing of names (methods, references or links) between different components. Moreover, it appears clear that this kind of adaptation can solve only renaming-based mismatchings of very similar behaviours. In general, one is interested in adapting a larger set of situations where, for instance, reordering and remembering of messages may be necessary.

We represent an adaptor specification by means of a mapping that establishes a number of rules relating actions and data of two components. For instance, the mapping expressing the intended adaptation for the previous example is trivially written as:

\[
M = \{ \text{request!}(\url) \leftrightarrow \text{query?}(\url); \\
\text{reply?}(\text{file}) \leftrightarrow \text{return!}(\text{file}); \}
\]

where each row is a rule of the mapping. The intended meaning of the first rule of \( M \) is that every time \( P \) will perform a \texttt{request!} output action, \( Q \) must eventually perform a corresponding \texttt{query?} input action. The use of parameters \texttt{url} and \texttt{file} in the mapping explicitly states the desired circulation of data among the two components. Indeed, in the previous mapping, \texttt{url} is not the actual datum sent by \( P \), but a parameter in the mapping stating the flow of information from the \texttt{request} made by \( P \) to the \texttt{query} expected by \( Q \). According to the name passing features of IP-calculus, parameters may also occur in place of references, so that for example, an adaptor can send a reference (e.g. a channel or a method name) and then interoperate accessing that reference. Each of such parametric actions occurring in a mapping is called a \textit{p-action}, while a p-action whose parameters have been (partially) instantiated with actual data of the components is called a \textit{p-action instance}. 
Definition 4.1 (Mapping) A mapping for two roles P and Q is a set of rules \( \{R_1, \ldots, R_n\} \). A rule is an expression of the form:

\[
\alpha_1, \ldots, \alpha_n \leftrightarrow \beta_1 \ldots \beta_m;
\]

where, for all \( i, j \), \( \alpha_i \) is a p-action (whose instances are actions of P), and \( \beta_j \) is a p-action (whose instances are actions of Q).

The action and data correspondence stated by a mapping \( \mathcal{M} \) relative to components P and Q is satisfied by a mapping \( \mathcal{A} \) if for each trace of \( \{P, \mathcal{A}, Q\} \), for each action \( \alpha\sigma \) (instance of a p-action \( \alpha \) in \( \mathcal{M} \)) executed in the trace, a rule \( R_i \) exists in \( \mathcal{M} \) such that all the actions in \( R_i\sigma \) have an instance which is executed in the trace.

Moreover, the adaptor can not “forge” the value of a mapping parameter and send it before it has been instantiated by means of an input action of the adaptor, which not necessarily belongs to the same mapping rule. An exception to this requirements is given by constant values that are explicitly specified in the mapping and allow the adaptor to send a value it has not previously received (some examples follow in the rest of the section).

Mapping rules state also correspondences between sent and received data. Parameters occurring in the p-actions have a global scope in the mapping and are instantiated in mapping output actions (which actually correspond to input actions of the adaptor). There are not consistency constraints like the closeness of rules, so that for example in a rule a parameter not bound by any of the actions in the rule might occur, and be possibly bound by actions occurring in different rules. Moreover, it is also possible to write a mapping where a parameter is bound by two or more rules: non-deterministically, the constructed adaptor will choose one of the possible alternatives so as to decide which action binds the parameter. Data dependencies may hence be induced between actions of the same rule or also between actions of different rules.

An adaptor can not interfere with the communication by forging data on the behalf of one of the components it is adapting, unless explicitly required by the mapping.

This loose definition of mapping aims to provide the mapping designer with an expressive means, avoiding having to deal with too many low level details, that will be fixed by the adaptor construction procedure. Obviously, mapping syntax permits a designer to also define unfeasible mappings, like for example a mapping with parameter not instantiated by any action, that will not lead to an adaptor.

Informally speaking, a mapping provides a minimal specification of an adaptor that will play the role of a “component-in-the-middle” between the two components P and Q. Considering again the above mentioned mapping, the behaviour interface of an adaptor satisfying it, for instance, is:

\[
\text{request?(url). query!(url). return?(file). reply!(file). 0}
\]
Observe that this adaptor will maintain the name spaces of P and Q separated and prevent the two from interacting with each other without its mediation. Observe also that the introduction of the adaptor to connect P and Q has the effect of changing their communication from synchronous to asynchronous. Indeed, the task of the adaptor is precisely to adapt P and Q together, not to act as a transparent communication medium between them.

It is important to note that the adaptor specification defined by a mapping abstracts away from many details of the components behaviours. The burden of dealing with these details is put on the (automatic) adaptor construction process, that will be described in the next section. In order to appreciate the expressiveness of the mapping language introduced, consider, for example, a component that wants to print several copies of different kinds of documents by calling different methods, e.g. `print_postscript!(d1)` and `print_pdf!(d2)`, and a driver that is able to print both the formats by the unique method `print?(x)`. The following mapping, where `doc` is actually bound by two different actions:

\[
M = \{ \text{print_postscript!(doc)} \Rightarrow \text{print?(doc)}; \\
\text{print_pdf!(doc)} \Rightarrow \text{print?(doc)}; \}
\]

expresses exactly the intended mapping without the need to take into consideration the actual protocols of the two components.

This section ends with some examples that illustrate the cases that can be modeled by the mapping specification language.

- **Multiple action correspondence.** While the previous examples dealt with one-to-one correspondences between actions, adaptation may in general require relating groups of actions of different components. For instance, consider two components P and Q involved in an authentication procedure. Suppose that P authenticates itself by sending first its user name and then a password. Q instead is ready to accept both the data in a single shot. Their behaviour interfaces and the mapping `M` specifying the required adaptation are:

  \[
  \text{user!(me). passwd!(pwd). 0 // P} \\
  \text{login?(usr, word). 0 // Q} \\
  \]

  \[
  M = \{\text{user!(me), passwd!(pwd)} \Leftrightarrow \text{login?(me,pwd)};\}
  \]

  The mapping associates both output actions performed by P to the single input action performed by Q, indicating also the reordering of parameters to be performed by the adaptor.

- **Actions without a correspondent.** Adaptation must also deal with situations in which an action of a component does not have a correspondent in the other component. For instance, consider a component P that authenticates itself
(actions \texttt{usr!} and \texttt{passwd!}), asks for the list of files which are present in a repository (\texttt{dir!} and \texttt{getdir?}), and then deletes a file (\texttt{delete!}). The repository server \textit{Q} does not require a login phase, but it rather expects a password to be sent together with the invocation of each service it provides (\texttt{ls?} for listing files, and \texttt{rm?} for deleting a file):

\begin{verbatim}
user!(me). passwd!(pwd). dir!(). getdir?(list). delete!(file). 0 // P
ls?(password). return!(files). rm?(name, password). 0 // Q
\end{verbatim}

From the viewpoint of \textit{Q}, authentication concerns are spread over the whole interaction. Moreover, notice that the parameter \texttt{me} is not requested while \texttt{pwd/passwd} is used more times by \textit{Q}.

In order to explicitly represent this conceptual asymmetry among the two components, and hence to facilitate the task of devising and reasoning about the high-level specification of a mapping, we have introduced the keyword \texttt{NONE}. The actions of a component which do not have a correspondence in the other component may be associated with \texttt{NONE}. Hence, the following mapping states that the login phase of \textit{P} has no correspondence in \textit{Q} and also that the parameter \texttt{pwd} must be recorded for subsequent uses.

\begin{verbatim}
M = { user!(me), passwd!(pwd) <> NONE;
    dir!() <> ls?(pwd);
    getdir?(files) <> return!(files);
    delete!(file) <> rm?(file, pwd); }
\end{verbatim}

- \textit{Constants and names in the mapping}. Sometimes, adaptation may be necessary also to overcome discrepant usage of data or to resolve conflicting usage of references. Consider for example two components exchanging data. One of them signals an error condition, say end of file, by the call to a specific error-managing procedure: \texttt{error_eof!()}. The other component instead, reacts to errors when they are signaled through the standard input channel, e.g. by the action \texttt{input?(error)}. The two components can be easily adapted, by simply knowing the code which is used by the second component to encode the end of file error, say EOF:

\begin{verbatim}
M = { error_eof!() <> input?(EOF); }
\end{verbatim}

In this case the adaptor is allowed by the mapping to forge a constant value for a component it has not previously received.

As far as references are concerned, imagine a component which needs a channel on which to send data. In case the other component is waiting to receive those data, but it is not able to provide a channel, the mapping specification may require that the adaptor itself provides the channel, acquires data and sends them to the other waiting component:
\[ M = \{ \text{NONE} \leftrightarrow \text{link?(new channel)}; \]
\[ \text{receive?(data)} \leftrightarrow \text{channel!(data);} \} \]

As well as for the case of constant, the adaptor can forge the channel indicated by the mapping. We assume that we are able to distinguishing, in a mapping, between parameters, constants (typically uppercase), and new names (typically preceded by the keyword `new`).

- **Non-deterministic associations between actions.** A difficult case for adaptation arises when the execution of a component action may correspond to different actions to be executed by the other component. Indeed, in general each component may perform local choices to decide what action to execute next. In such cases, adaptation should take care of dealing with many possible combinations of actions independently performed by the two components.

In order to feature a high-level style of the specification of the desired adaptation, we allow non-determinism in the adaptor specification. For instance, suppose a component receiving a file by means of a single action `read?` while its counterpart may decide to send an action `data!(x)` or an end-of-file `eof!()`. The mapping will be specified by means of two separate rules:

\[ M = \{ \text{read?(x)} \leftrightarrow \text{data!(x)}; \]
\[ \text{read?(EOF)} \leftrightarrow \text{eof!();} \} \]

The adaptor derivation process will then be in charge of building an actual adaptor capable of dealing with all the possible specified situations. Once more, our goal is to allow the adaptor specification to abstract away from implementation details, and to leave the burden of dealing with these details to the (automatic) adaptor construction process. The use of non-deterministic associations will be illustrated further in the example of Section 4.5.

### 4.4 Adaptor derivation

Given two roles \( P \) and \( Q \) and a corresponding mapping \( M \), the generation of an adaptor, if there is one, is achieved by an automatic procedure that is a variant of the \( A \) algorithm for acceptability of Figure 3.8.

The goal of the algorithm is to build a process \( A \) such that:

1. \( \{P,A,Q\} \) is successful (i.e. all traces lead to a successful state, where both \( P \), \( Q \) and \( A \) have reduced to 0), as for completions, and

2. \( A \) satisfies the given mapping \( M \), that is, all the action correspondences and data dependencies specified by \( M \) are respected in any trace of \( \{P,A,Q\} \).
The algorithm incrementally builds the adaptor $A$ by trying to eliminate progressively all the possible deadlocks that may occur in the evolution of $\{P, A, Q\}$. The strategy with which the state space is explored is the same used by $\mathcal{A}$ algorithm, while the choice of the action with which to expand the incrementally built adaptor needs to conform to the mapping. Three main changes have been necessary in order to exploit the $\mathcal{A}$ algorithm for adaptor construction. In particular, it has been necessary i) to introduce a set $R$ of the actions “activated” throughout a trace, i.e. the actions that according to mapping rules must be executed before the trace ends, ii) to introduce a set $P$ of parameter bindings, and iii) to modify the definition of the $\textit{triggers}$ function.

When a component synchronises with the adaptor by means of an action occurring in the mapping, the parameters of the mapping action are instantiated coherently with the values possibly communicated by the component. Since, as explained, parameters have a global scope in the mapping, it is necessary to record the binding parameter-value, so that it can be used in case the parameter occurs again, possibly in a different rule. Such bindings are recorded in the set $P$.

Consider, for instance, the case of a component sending its login name and password to a server which only needs the login name to authenticate the component, while it requires the password for any successive request. The following mapping specifies the required adaptation:

$$M = \{ \text{user!(me, pwd) <> id?(me);}$$
$$\text{request!(command) <> execute?(command, pwd);} \}$$

The execution of the component action $\text{user!(alice, cat)}$ requires, according to the mapping, the execution of the server action $\text{id?(alice)}$. Moreover, a side effect of the communication is that the adaptor “knows” that the value $\text{cat}$ is associated to the parameter $\text{pwd}$, by means of a binding stored in $P$, for future communications.

We resume the functioning of the algorithm, which is correspondent to that of the algorithm $\mathcal{A}$. The first step is to define which are the triggers that can be selected to trigger a deadlocked state, respecting the definition of the mapping $M$. Three cases must be distinguished:

1. there exists a triggering action, according to the standard definition of $\textit{trigger}$ with no corresponding p-action in any rule of the mapping. In this case no adaptation is required and the adaptor behaves as a standard completion.

2. there exists a triggering component action $\alpha$ and a corresponding p-action $\alpha'$, occurring in the mapping, such that an assignment makes them equal. For instance, referring to the above example, the action $\text{user!(alice, cat)}$ is made equal to the p-action $\text{user!(me, pwd)}$, occurring in the first rule of the mapping, by the parameter assignment $\Delta = [\text{me} \rightarrow \text{alice}, \text{pwd} \rightarrow \text{cat}]$. Such an assignment can
• instantiate variables of the component action with values sent by the adaptor, i.e. constants and new names occurring in the mapping actions, or values acquired in previous communications according to the binding set \( \mathcal{P} \),

• instantiate parameters of the mapping action by means of already known bindings belonging to \( \mathcal{P} \),

• instantiate parameters of the mapping action with values sent by one of the two components, possibly acquiring new parameter bindings, which are recorded \( \mathcal{P} \).

The mapping action \( \alpha' \) non-deterministically selects one of the rules in which it occurs. All the other actions in the rule, possibly partially instantiated according to the parameter bindings, are activated and recorded in \( \mathcal{R} \): they must be eventually executed, before the current trace ends. The adaptor \( A \) is expanded with an action \( \alpha \), like \( \text{user?(me, pwd)} \) in the example, which is dual to the component action (as happens in the case of completion construction).

3. a triggering component action \( \alpha \) has a corresponding p-action already activated, i.e. exists \( \alpha' \in \mathcal{R} \), such that \( \alpha = \alpha' \) up to an appropriate mapping \( \Delta \). Then, \( \alpha' \) is consumed from \( \mathcal{R} \), \( \alpha' \) used to expand \( A \) and the bindings in \( \mathcal{P} \) are appropriately updated with the possibly new ones acquired in the communication. No action is activated.

Note that case 2 and 3 may overlap: a triggering action can either activate a new rule or consume an action previously activated. This is reflected in the algorithm in a non-deterministic choice, which eventually explores all the possibilities.

The new definition of triggering action is given by the function \( M\text{triggers}(\mathcal{N}, \kappa, \mathcal{M}, \mathcal{R}, \mathcal{P}) \), where \( \mathcal{N} \) and \( \kappa \) are, like in \( A \) algorithm, a set of nodes and the adaptor (completion) knowledge, \( \mathcal{M} \) is the mapping, \( \mathcal{R} \) a set of activated (p-)actions, and \( \mathcal{P} \) the set of parameter bindings. According to the above mentioned possibilities for the existence of a trigger, the function returns a set of tuples \( (\beta, \Delta, \text{Rule actions}) \), where \( \beta \) is a trigger for one of the states in \( \mathcal{N} \) which respects \( \mathcal{M} \), \( \Delta \) is an assignment (to the parameters of the mapping), and \( \text{Rule actions} \) is the possibly empty set of rule actions activated by the choice for \( \beta \).

\[
M\text{triggers}(\mathcal{N}, \kappa, \mathcal{M}, \mathcal{R}, \mathcal{P}) = \{ (\alpha, \epsilon, \emptyset) \mid \exists \{P_1, A, P_2\} \notin \mathcal{N}. \ P_i \xrightarrow{\alpha} P_i' \text{ (with } i = 1 \lor i = 2) \land \alpha \notin \mathcal{M} \land \text{ref}(\alpha) \in \kappa \} \]

\[
\cup \{ (\alpha, \Delta|_\mathcal{M}, (\text{Rule})\Delta|_\mathcal{M}) \mid \exists \{P_1, A, P_2\} \notin \mathcal{N}. \ P_i \xrightarrow{\beta} P_i' \land \alpha \in \text{Rule} \in \mathcal{M} \land \text{ref}(\alpha) \in \kappa \land \exists \Delta : \alpha\Delta = G \beta\Delta \}
\]

\[
\cup \{ (\alpha, \Delta|_\mathcal{M}, \emptyset) \mid \exists \{P_1, A, P_2\} \notin \mathcal{N}. \ P_i \xrightarrow{\beta} P_i' \land \alpha \in \mathcal{R} \land \text{ref}(\alpha) \in \kappa \land \exists \Delta : \alpha\Delta = G \beta\Delta \}
\]
The first set deals with case 1, it is analogous to the trigger definition for the A algorithm, where $\alpha \notin M$ is read as “there is not a p-action $\alpha'$ in any of the rules of $M$ such that $\tilde{\alpha}$ is an instance of it”. This case returns an empty assignment and an empty set of activated actions. According to the new syntax adopted, the knowledge of the channel ($ch(\alpha) \in \kappa$) is now the knowledge of the reference (e.g. method name) of $\alpha$ ($ref(\alpha) \in \kappa$).

The second set deals with case 2. If a triggering action $\beta$ is offered by one of the two components, and $\alpha$ is an action in a rule of the mapping, whose reference is known, and that can be made equal to $\beta$ by an assignment $\Delta$, then action $\alpha$ activates the rule Rule of $M$ to which it belongs. Strictly speaking, the assignment must make the two actions equal and ground ($\approx_G$) so that communication can happen (e.g. the adaptor can not send a not instantiated parameter) according to the semantics of communication (see Figure 3.5). Action $\tilde{\alpha}$ is returned as trigger, together with the assignment restricted to the parameters of the mapping $\Delta|M$. The selection of the action $\alpha \in Rule$ activates the remaining actions in Rule, which are returned, in their dual form, in order to be added to $R$.

The third set deals with case 3. It is analogous to the previous case, but, the action of the component corresponds to an already activated action ($\alpha \in R$). The reference of the action is known, and the assignment $\Delta$ makes $\alpha$ and $\beta$ equal. The triggering action $\tilde{\alpha}$ is returned in order to be removed from $R$, together with the parameter assignment and no activated action.

The algorithm $ADAPT$ is presented in Figure 4.1. Its structure is extremely close to that of the A algorithm, from which it derives. We will explain the main features and differences of $ADAPT$ with respect to $A$, referring to the explanation of $A$ (Figure 3.8) for further details.

The input of the algorithm consists of the behavioural expressions of two roles of components that must be adapted and a corresponding mapping. Initially, the adaptor knows all the references in the components: each component offers to the adaptor all its methods. The set $R$, that records the actions that must (still) be done according to $M$, is initialised as the empty set, as well as the set $P$ containing the known assignment for the mapping parameters.

If the components do not need adaptation, the empty adaptor is returned. Otherwise, a triggering action is chosen according to the new definition of $M_{\text{triggers}}$. If there are no such triggers, the components are not adaptable. Note that the definition of $M_{\text{triggers}}$ takes into consideration all the triggers that the function triggers was considering, but some of them may be discharged because they are triggers that do not respect the extra data dependencies induced by the mapping.

---

1It is worth remembering that the assignment $\Delta$ may instantiate parameters according to the set of known bindings $P$, which hence is a necessary parameter for the function $M_{\text{triggers}}$, on which $\Delta$ depends.

2For instance, referring to the previous example, the assignment $[pwd \rightarrow \text{Cat}] \in P$ can be used to instantiate the parameter $pwd$ of action $\text{execute?}(\text{command},pwd)$. 

---
$\text{ADAPT}(P_1, P_2, M)$

\[
\begin{align*}
\kappa &= \text{ref}(P_1 \cup P_2); \\
\mathcal{R} &= \emptyset; \quad \mathcal{P} = \emptyset; \\
\text{if} \ \text{deadlocks}(\text{nodes}([P_1, P_2])) = \emptyset \\
&\quad \text{then return}([]) \quad \text{// S totally correct} \\
\text{else} \{ \\
&\quad \mathcal{TR} = M_{\text{triggers}}(\text{nodes}([P_1, P_2]), \kappa, M, \mathcal{R}, \mathcal{P}); \\
&\quad \text{if} \ \mathcal{TR} = \emptyset \quad \text{// permanent deadlock} \\
&\quad \quad \text{FAIL} \\
&\quad \text{else} \{ \\
&\quad & (\beta, \Delta, \text{Rule\_actions}) = \text{CHOICE}(\mathcal{TR}); \quad \text{// build P} \\
&\quad & A = \beta; \\
&\quad & \kappa = \text{update}(\kappa, \beta); \\
&\quad & \mathcal{R} = \mathcal{R} \cup \text{Rule\_actions} \setminus \beta; \quad \mathcal{P} = \mathcal{P} \cup \Delta; \\
&\quad & \text{while} \ \text{deadlocks}(\text{nodes}([P_1, A, P_2])) \neq \emptyset \\
&\quad & \quad \text{do} \{ \\
&\quad & & \mathcal{N} = \text{evolutions\_after}([P_1, A, P_2], \text{last}); \\
&\quad & & \text{if} \ \text{deadlocks}(\mathcal{N}) \neq \emptyset \land \text{successes}(\mathcal{N}) \neq \emptyset \quad \text{// permanent deadlock} \\
&\quad & & \quad \text{then FAIL} \\
&\quad & & \quad \text{else if} \ \text{deadlocks}(\mathcal{N}) = \emptyset \quad \text{// so far so good:} \\
&\quad & & \quad \quad \text{undo}(\kappa, \text{last}); \quad \text{undo}(\mathcal{R}, \beta, \text{Rule\_actions}); \\
&\quad & & \quad \quad \text{undo}(\mathcal{P}, \Delta) \\
&\quad & \quad \text{else} \{ \\
&\quad & & \mathcal{TR} = M_{\text{triggers}}(\text{nodes}([P_1, A, P_2]), \kappa, M, \mathcal{R}, \mathcal{P}); \\
&\quad & & \text{if} \ \mathcal{TR} = \emptyset \quad \text{// permanent deadlock} \\
&\quad & & \quad \text{then FAIL} \\
&\quad & & \text{else} \{ \\
&\quad & & & (\beta, \Delta, \text{Rule\_actions}) = \text{CHOICE}(\mathcal{TR}); \quad \text{// build P} \\
&\quad & & & A = \text{add} \beta \text{ to } A \text{ as a new child of last}; \quad \text{undo}(\kappa, \text{last}); \quad \text{undo}(\mathcal{R}, \beta, \text{Rule\_actions}); \\
&\quad & & & \text{undo}(\mathcal{P}, \Delta); \quad \text{undo}(\mathcal{R}, \Delta) \\
&\quad & & \} \\
&\quad & \} \\
&\quad \text{if } \mathcal{R} = \emptyset \text{ then return}(A) \text{ else FAIL}; \\
&\} \\
&\}
\end{align*}
\]

Figure 4.1: Adaptor construction.
The adaptor is initially expanded according to the chosen trigger and the knowledge is updated, as usual (where references are updated instead of channel names).

The set of activated rules is updated according to the set \textit{Rule\_actions} returned by \textit{M\_triggers}. If the set is empty, then either the trigger does not regards the mapping (and hence the trigger $\beta$ can not be in $\mathcal{R}$) and $\mathcal{R}$ is not modified, or the trigger consumes an action already in $\mathcal{R}$, that hence is removed form it. If the set is not empty, a new rule has non-deterministically been activated by the trigger, and all its actions but the trigger itself are inserted in $\mathcal{R}$.$^3$

The set $\mathcal{P}$ is updated with the parameter assignments possibly determined by the triggering action.

The while cycle works exactly as the while cycle of the $\mathcal{A}$ algorithm, except for the managing of activated actions $\mathcal{R}$, and also for the necessary updating of the data structure when backtracking ($\text{undo}(\mathcal{R}, \beta, \text{Rule\_actions})$, $\text{undo}(\mathcal{P}, \Delta)$).

Non-determinism is dealt with in the same manner as it is dealt with in the $\mathcal{A}$ algorithm, as well as for failure and termination. The algorithm successfully terminates when one of its instances successfully terminates since the derivation tree of $\{ \mathcal{P}_1, \mathcal{A}, \mathcal{P}_2 \}$ does not contain deadlocks and $\mathcal{A}$ satisfies the mapping.

Differently from acceptability checking, where the values of the transmitted data do not play a relevant part, the process of adaptation is committed to properly delivering data according to a mapping. It is hence possible that even if two components can be completed they can not be adapted. This is reflected in the algorithm, by the extra constraints imposed on the actions of the completion/adaptor. In particular, the definition of \textit{M\_triggers} restricts the possible triggers according to the constraints over parameters, since, because of the assignment $\Delta$, an adaptor can perform an action only if it has previously acquired the needed parameters. Moreover, the construction of an adaptor, even if it respects parameter constraints, may fail because of the constraints due to the rules of the mapping, i.e. the correspondence among action instances, as happens, for instance, if at the end of the construction some actions still belong to $\mathcal{R}$, the set of activated rules.

Hence, as expected, adaptation may fail more easily than acceptability. Clearly, failures due to the newly introduced procedures, like \textit{M\_triggers} and the managing of the activated rules, correspond to instances of the adaptor construction that would not have led to an adaptor satisfying the mapping. The exhaustive non-deterministic search, analogously to the case of the $\mathcal{A}$ algorithm, guarantees that the existence of an adaptor implies the success of at least an instance of the algorithm.

Considering again the example relative to the point \textit{Actions without a correspondent} of Section 4.3 about a file repository server, the algorithm constructs the following adaptor $\mathcal{A}$:

\footnote{More properly, $\mathcal{R}$ is a multi-set of actions, containing more instances of the same action possibly activated.}

It is easy to verify that the composition \{P,A,Q\} is deadlock free, and that A satisfies the mapping, both in terms of action correspondence and data dependencies (e.g., A forwards pwd and file to Q only after receiving them from P).

### 4.5 An example of adaptation

Consider a typical FTP transmission in which a file is sent by a server to a client. The example is simplified to show only the relevant details, while hopefully keeping its realistic flavour.

In order to make a modular specification of the problem, we will consider two different interactions between the client and the server, using two roles for describing their behavior. First, we will describe the behaviour of the client and the server regarding how to create and close an FTP session, and how to request the typical put and get services for transmitting a file. Second, we will describe the more low-level details of file transmission using a separate pair of roles. Two adaptors will be produced, one for each two role-to-role connections specified by a different mapping.

The roles IServer and IClient describe the interface of the server and the client regarding the use of FTP commands.

```plaintext
class IServer = 
  interface open?(Link ctrl);
  user?(Data name, Data password);
  put?(Data filename, Link ctrl);
  get?(Data filename, Link ctrl);
  close?(Link ctrl);

  behaviour open?(ctrl).
      user?(name,password,ctrl).
      ( put?(filename,ctrl). close?(ctrl). 0
        + get?(filename,ctrl). close?(ctrl). 0
        + close?(ctrl).0 )
```

Each service offered by the role IServer requires a socket, i.e. the link ctrl, by means of which the service will be provided. Note that, because of having partitioned the behaviour of the server into more roles, this link does not support any communication in this role, but it is anyway a datum present in its behaviour. Each instance of the role for the low-level file transmission exploits such a link to establish a separate connection with the client. Even if for ease of presentation the two links have the same name (and are the same link in the server application), our framework does not permit roles to share a state, being oriented to the verification of local properties, and hence the two are distinct links in different roles.

An FTP session is opened after an action open, after which, the client must identify itself with a name and password. Then, put and get commands for file
uploading and downloading can be issued to the server. Finally, the connection is ended with close.

role IClient = {
    interface login!(Data usr);
    pass!(Data pwd);
    getfile!(Data file);
    logout!();

    behaviour login!(usr).
    pass!(pwd).
    getfile!(file).
    logout!().
}

The role IClient, instead, specifies that the client will connect with a login message by which it sends its identity, followed by its password in a separate message (however no control socket is provided). Then, the client asks for a certain file, and finally logs out.

It is worth observing that, in spite of the different behaviours of the two components, their adaptation can be simply specified by the mapping:

\[ M = \{ \text{login!(usr), pass!(pwd) } \leftrightarrow \text{open!(new ctrl), user!(usr,pwd,ctrl);} \]
\[ \text{getfile!(file) } \leftrightarrow \text{get!(file,ctrl);} \]
\[ \text{logout!()} \leftrightarrow \text{close!(ctrl);} \}

The first rule of \( M \) establishes the intended correspondence between two pairs of actions of the components. The mapping also exploits the use of action parameters to specify data dependencies among different actions of the components. In particular the ctrl parameter is employed in all the three rules to specify the needed adaptation due to the fact that the client does not specify the control socket in its protocol. The special keyword \text{new} in the first rule is used to specify the need for the adaptor to create a new name to match the server open? input action. As shown in Section 4.4, this mapping will produce the adaptor:

\[ A = \text{login?(usr). pass?(pwd). (ctrl) open!(ctrl). user!(usr,pwd,ctrl).} \]
\[ \text{getfile?(file). get!(file,ctrl). logout?(). close!(ctrl).} \]

which allows both components to interact successfully.

Let us consider now how a file is transmitted once a get command is issued by the client. Typically, the server will create a separate thread (daemon) for the transmission of the file. Accordingly, we will use another pair of roles, namely IGetDaemon and IGettingFile, to represent this facet of the interaction between client and server, respectively:

role IGettingFile = {
    interface read?(Data x);
    break!();
}
The differences between the two roles are the following:

- Server action \( \text{ctrl!} \) does not have a correspondent in the client, reflecting the fact that while the server creates specific links for each file transmission, the client uses fixed, predefined links for the same purpose. Hence, a suitable mapping rule for this situation is:

\[
\text{none} \leftrightarrow \text{ctrl!(data, eof)};
\]

- The action for reading each piece of the file is called \( \text{read?} \) in the client, while the corresponding action in the server is \( \text{data!} \). This mismatch can be easily solved with the mapping rule:

\[
\text{read?(x)} \leftrightarrow \text{data!(x)};
\]

- The server may indicate at any moment the end of the file being transmitted by sending an \( \text{eof!()} \), but the client does not have a corresponding message. This situation can be dealt with by using a special value in message \( \text{read}? \), thus allowing the client protocol to end:

\[
\text{read?(EOF)} \leftrightarrow \text{eof!()};
\]

- The client can autonomously decide (because of its local choice) to break the transmission at any moment by sending a \( \text{break!()} \) message. This case is more difficult to adapt, since the server and the adaptor might have already engaged themselves in a pair of complementary \( \text{data!} \) actions. This would violate the one-to-one correspondence between actions \( \text{read?} \) and \( \text{data!} \) expressed by the second rule of the mapping (and no adaptor could be produced). We can solve this problem by including in the mapping rules for the cases in which a client sends a break, while the server is sending the file to the adaptor, letting the adaptor solve the mismatch by “absorbing” both the communications. The client \( \text{break!()} \) is hence mapped to the actions \( \text{read!} \) and \( \text{eof!} \) of the server. Hence, the mapping is:
4.6. ABOUT THE ROLE OF THE ADAPTOR

\[
M = \{ \text{none }\leftrightarrow \text{ctrl!(data, eof);} \\
\quad \text{read?(x) }\leftrightarrow \text{data!(x);} \\
\quad \text{read?(EOF) }\leftrightarrow \text{eof!();} \\
\quad \text{break!() }\leftrightarrow \text{data!(y);} \\
\quad \text{break!() }\leftrightarrow \text{eof!();} \}
\]

Notice that the above mapping specifies action correspondences in a non-
deterministic way. For instance, the last two rules state that the execution of the
break! action may correspond to either a data! action or to a eof! action on
the server side. Similarly, the second and fourth rule specify that the execution of
a data! output operation by the server may match either a read? or a break!
operation autonomously performed by the client.

It is important to observe that allowing non-deterministic correspondences in the
mapping features a high-level style of the specification of the desired adaptation.
While the mapping simply lists a number of possible action correspondences that
may arise at run-time, the non-deterministic adaptor derivation process will be in
charge of devising the actual adaptor able to suitably deal with all the possible
specified situations. The adaptor produced from the above mapping is:

\[
A = \text{ctrl?(data, eof). ( data?(x). ( read!(x). 0 + break?(). 0 ) + eof?(). ( read!(EOF). 0 + break?(). 0 ) + break?(). ( data?(x). 0 + eof?(). 0 ) )}
\]

4.6 About the role of the adaptor

In this section we discuss how to enhance the mapping specification language so as
to provide the adaptor designer with a finer control over the constructed adaptors.

The task of an adaptor is to facilitate interoperation among components accord-
ing to the provided mapping. While there can be different ways in which the adaptor
can accomplish its task, the automatic adaptor generation procedure is orientated
to produce an adaptor which has the “minimal” influence necessary to let the com-
ponent interoperate, e.g. an adaptor is not allowed to forge values not specified in
the mapping. Moreover, the mapping language has been defined as abstractly as
possible, since it is intended as an high-level support for the mapping specifier, that
avoids to him or her the necessity of providing too many low-level details.

For example, the mapping does not specify an ordering among the actions of a
rule, nor among different rules, allowing the algorithm the task (and the freedom)
to define one order which is compatible with the successful termination and with the
other constraints imposed by the mapping. If, on the one hand, this reveals a lack
of a finer control over the protocol of the adaptor (and hence of the components), on
the other hand it avoids the mapping designer the error-prone and time consuming
problem of fixing a protocol, while the algorithm will do it for him or her.
Nonetheless, a more precise control of the way in which the adaptor manages the data and the actions of the components can be easily embedded in the proposed framework.

The adaptor, “man in the middle”, works as a buffer of all the data exchanged by the components, and it is able to manipulate the order in which data are received, to forget some data not required or to remember and to repeatedly provide data acquired from the components.

Consider, for instance, two components \( P \) and \( Q \) exchanging a sequence of two numbers. Suppose that their corresponding interaction patterns are:

\[
P = \text{write!}(n_1). \text{write!}(n_2). \ 0 \\
Q = \text{read?}(m_1). \text{read?}(m_2). \ 0
\]

A natural mapping is:

\[
M = \{ \text{write!}(n) <> \text{read?}(n); \}
\]

However, it is worth observing that, because of the asynchronous nature of the adaptor, data may not be received in the correct order, as in the case of the following adaptor, which fulfills mapping requirements:

\[
A = \text{write?}(n_1). \text{write?}(n_2). \text{read!}(n_2). \text{read!}(n_1). \ 0
\]

In this case the adaptor is constructed by activating the mapping rule twice as a consequence of triggering the two output actions of \( P \), and then by consuming the activated rules \( \text{read!}(n_2) \) and \( \text{read!}(n_1) \), letting \( Q \) terminate.

Consider now another example further illustrating the effects that the adaptor may cause on the interaction of components. A component \( Q \) provides spell-checking and printing facilities. It requires an identification of users who have access to the services and a payment for every service provided. Component \( P \), instead, accesses the services by sending its name and a payment and then requires the services:

\[
P = \text{access!}(Alice, 1000). \text{correct!}(thesis). \text{print!}(thesis). \ 0 \\
Q = \text{identify?}(user). \\
\quad (\text{print?}(document, money). \ 0 \ || \ \text{spell?}(document, money). \ 0)
\]

A natural way to conceive the mapping is:

\[
M = \{ \text{access!}(user, euro) <> \text{identify?}(user); \\
\quad \text{correct!}(doc) <> \text{spell?}(doc, euro); \\
\quad \text{print!}(doc) <> \text{print?}(doc, euro); \}
\]

from which the following adaptor is generated:

\[
\text{access?}(user, euro). \ \text{identify!}(user). \ \text{correct?}(doc). \\
\text{spell!}(doc, euro). \ \text{print?(doc). \ \text{print!}(doc, euro). \ 0}
\]
Obviously, even if it fulfills the mapping, this adaptor does not implement the intended adaptation, since it lets $P$ use two services while it pays only one of them.

This erroneous outcome of the adapting process depends on the capacity of the adaptor to “remember” the data transmitted by the components for future communications, e.g. the parameter `euro`. Also the inverted sequence problem of the previous example is due to the capability of the adaptor to remember data, together with the possibility of activating more rules concurrently.

A mapping specifier could realise that the two components can not be adapted, by analysing the actual protocol of $P$ so to discover that it requires two services and issues only one payment. In order to avoid the, in general, costly analysis of protocols, one could require that adaptors can not “remember” parameters, but neither this would be a solution. In fact, in some cases like in the example relative to the point *Actions without a correspondent* of Section 4.3, the capability of the adaptor to remember data, the password of $P$ in the example, permits the two components to be successfully adapted.

The problem of controlling the adaptor’s data usage has been taken into consideration in many of the studies about adaptation and it was present also in the seminal paper [YS97], where it was addressed by providing a means to check the usage of data *after* the adaptor construction.

In our framework, relying on the data structures employed by the algorithm, it is possible to enhance the mapping language so as to specify how the adaptor is required to deal with data replication and mapping rule concurrent activations.

Both rules and data, in terms of parameter bindings, are managed in the algorithm as sets, $R$ and $P$, i.e. unordered structures. Ordering such structures is a simple way to address problems like the ones previously illustrated. In particular the following policies can be adopted:

- The set $R$ is provided with a FIFO (first-in, first-out) discipline, as far as multiple activations of the same p-action are concerned.

  This is simply implemented by reading the condition $\alpha \in R$ in the definition of $M_{\text{trigger}}$ as: “select the first occurrence of $\alpha$ in $R$, according to a FIFO discipline”. In other words, the chosen $\alpha$ is the first of such actions that has been activated.

- The set $P$ is provided with a “destructive reading”, i.e. every time a parameter binding occurring in $P$ is used to match a triggering component action, then the binding is removed from $P$.

  This means that in the construction of $\Delta$ in the definition of $M_{\text{trigger}}$ it is not possible to use a binding in $P$ already used while triggering a previous action, unless the binding has been newly acquired.

Since, as already said, a statically adopted policy would not fit the generality of the situations occurring in adapting components, the illustrated policies need to be specified case by case within the mapping.
As an example, let us consider again the previous example about printing and spelling charged services. Let us assume that the default policy is the one presented in the algorithm. Let us also assume that the destructive reading policy can be required for a parameter by labeling the parameter occurrence with inverted commas. The mapping then becomes:

\[ M = \{ \text{access!(me, euro)} \leftrightarrow \text{identify?(me)}; \]
\[ \text{correct!(doc)} \leftrightarrow \text{spell?(doc, 'euro')}; \]
\[ \text{print!(doc)} \leftrightarrow \text{print?(doc, 'euro')} \} \]

which explicitly declares that every service must be paid for. After the identification phase and the first component request, the adaptor construction process yields:

\[ A = \text{access?(me, euro)} \cdot \text{identify!(me)} \cdot \text{correct?(doc)}. \]
\[ R = \{ \text{spell!(doc, 'euro')} \} \]
\[ P = \{ \text{me} \rightarrow \text{Alice}, \text{euro} \rightarrow 1000, \text{doc} \rightarrow \text{thesis} \} \]

An instance of the next iteration of the algorithm triggers the action \( \text{spell!(doc, euro)} \), and, in doing so it uses and consumes the binding \( [\text{euro} \rightarrow 1000] \) from \( P \), preventing the next action \( \text{print!(document, money)} \) of the service provider to be triggered, because the adaptor can not instantiate the parameter \( \text{euro} \) in the dual action \( \text{print!(doc, euro)} \), since the binding has been removed from \( P \).

In the same manner rules can be labeled with the FIFO policy. More in general, policies represent a modular way of specialising the algorithm. Indeed, they can be defined and plugged in the algorithm according to the user needs. In this way, adaptor properties can be enforced during adaptor construction rather than checked after it.

4.7 Concluding remarks

Several authors have proposed to extend current IDLs in order to deal with behavioural aspects of component interfaces. The use of FSMs to describe the behaviour of software components is proposed for instance in [CMK98, MKG99, Nie95, YS97]. The main advantage of FSMs is that they support a simple and efficient verification of protocol compatibility. On the other hand, such a simplicity is a severe expressiveness bound for modelling complex open distributed systems.

Process algebras feature more expressive descriptions of protocols, enable more sophisticated analysis of concurrent systems [AG97, MKM99, NNS99], and support system simulation and formal derivation of safety and liveness properties. In particular, the \( \pi \)-calculus, differently from FSMs and other algebras like CCS, can model some relevant features for component-based open systems, like dynamic creation of new processes, dynamic reorganization of network topology (mobility), and local and
global choices. The usefulness of $\pi$-calculus has been illustrated for describing component models like COM [Fei99] and CORBA [GZ99], and architecture description languages like Darwin [MEK95] and LEDA [CPT99a].

However, the main drawback of using process algebras for software specification is related to the inherent complexity of the analysis, and its possible undecidability.

In order to manage complexity, the IP-calculus appears a natural candidate by means of which developing more expressive adaptor specifications than those based on FSM, following our previous work on the use of modular and partial specifications by projecting behaviour over time (finite interaction patterns), and similarly to other approaches projecting it, for example, over space (roles) [CPT01]. Moreover, it must be mentioned that the finiteness of the state space generated by IP-calculus patterns does not give rise to decidability problems.

A general discussion of the issues of component interconnection, mismatch and adaptation is reported in [Bos97, DR97, GAO95], while formal approaches to detecting interaction mismatches are presented for instance in [AG97, CPT01, CIW99]. The problem of software adaptation was specifically addressed by the work of Yellin and Strom [YS97], which constitutes the starting point for our work. They use finite state grammars to specify interaction protocols between components, to define a relation of compatibility, and to address the task of (semi)automatic adaptor generation. Some significant limitations of their approach are related to the expressiveness of the notation used. For instance, there is no possibility of representing local choices, parallel composition of behaviours, or the creation of new processes. Furthermore, the architecture of the systems being described is static, and they do not deal with issues such as reorganizing the communication topology of systems. All these possibilities immediately become available when using a $\pi$-calculus based language. In addition, the asymmetric meaning they give to input and output actions makes the use of \textit{ex machina} arbitrators for controlling system evolution necessary.

Another closely related work is [Reu01], where the extension of interfaces with FSMs is proposed in order to check the correct composition of, and also to adapt, non-compatible components. Protocols are divided into two views: the services the component offers, and those it requires from its environment. One limitation of the work is that these two views must be orthogonal, i.e. each time a service is invoked in a component it results the same sequence of external invocations, while this may usually depend on the internal state of the component. It should also be noticed that in that work, there is no indication of action signs, and only method invocation is represented, while our approach involves a more general setting in which any dialogue or protocol between components can be specified. Finally, adaptation is considered in that work only as restriction of behaviour; if the environment does not offer all the resources required, the component is restricted to offer a subset of its services, but no other forms of adaptation (like name translation, or treatment of protocol mismatch) is considered.
The main aim of our approach is to contribute to the definition of a methodology for the automatic development of adaptors capable of solving behavioural mismatches between heterogeneous interacting components.

The work done using IP-calculus as basic language for adaptor specification and construction has permitted us

- to provide a framework that, without presenting the complexity of the analysis of a full fledged process algebra, maintains some of its distinguishing features with respect to FSM, and, in particular, concerning the dynamical aspects such as network reconfiguration, that can not be addressed by FMS,

- to devise an expressive and conceptually simple language that facilitates the “declarative” definition of mapping, leaving out of consideration the exact and complete procedural details of the protocols of the components to be adapted. This permits our specifications to be partial, differently, for instance, from the ones of [YS97], which closely correspond to the actual implementation of the full adaptor component,

- to exploit the state exploration techniques, developed for acceptability, for the automatic translation of a mapping specification into an actual adapting component, showing its generality as exploration strategy.

Our work falls in the research stream that advocates the application of formal methods, in particular of process algebras, to describe the interactive behaviour of software systems. As shown for instance in [BBT01a, CPT01], the adoption of π-calculus to extend component interfaces paves the way for the automatic verification of properties of interacting systems, such as the compatibility of the protocols followed by the components of the system.

While the proposed methodology lays a foundation for the automatic development of adaptors, we foresee several interesting further developments. The first we intend to address is the formal verification of properties of the generated adaptor, such as security properties, as suggested in [MKG99, YS97]. In practice, such a verification would allow an application to check that its security policy will not be spoiled by the inclusion of a new (adapted) component.

Moreover, the characterisation of a representative of the class of returned adaptors, or completions, is currently under study. The main difficulties are due to the fact that adaptors for the same components may be structurally very different.

Finally, the problem of mapping specification must be mentioned. As initially claimed, mapping design seems to necessarily require, in the general case, intelligent (human) supervision. Consequently, our aim in this dissertation has not been to address mapping design as such, but rather to provide an abstract notation intended to facilitate mapping specification by human designers, by abstracting away adaptor construction details from the process of mapping specification. Nonetheless, the study of conditions allowing for automatic, possibly run-time, mapping design (and, hence, adaptation) is an interesting direction for future research.
Chapter 5

Secure interaction in open systems

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Cryptography + Security protocols = Security?</td>
<td>123</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Cryptography</td>
<td>124</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Security protocols and their verification</td>
<td>125</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Security properties</td>
<td>128</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Discussion: (c)IP-calculus and security</td>
<td>129</td>
</tr>
<tr>
<td>5.2</td>
<td>The Intruder model</td>
<td>130</td>
</tr>
<tr>
<td>5.3</td>
<td>cIP: A Cryptographic calculus of Interacting Principals</td>
<td>137</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Syntax</td>
<td>138</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Sessions and Multiple-sessions</td>
<td>141</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Operational semantics</td>
<td>143</td>
</tr>
<tr>
<td>5.4</td>
<td>PL Logic: expressing security properties</td>
<td>150</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Syntax</td>
<td>151</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Models for PL</td>
<td>153</td>
</tr>
<tr>
<td>5.5</td>
<td>Concrete protocol verification</td>
<td>155</td>
</tr>
<tr>
<td>5.6</td>
<td>An Oracle discovers an attack</td>
<td>158</td>
</tr>
<tr>
<td>5.7</td>
<td>Symbolic protocol verification</td>
<td>162</td>
</tr>
<tr>
<td>5.7.1</td>
<td>A symbolic extension of cIP-calculus</td>
<td>164</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Correctness of symbolic protocol verification</td>
<td>167</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Generating symbolic messages</td>
<td>168</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Symbolic trace generation: An example</td>
<td>174</td>
</tr>
<tr>
<td>5.7.5</td>
<td>Symbolic verification</td>
<td>176</td>
</tr>
<tr>
<td>5.7.6</td>
<td>Symbolic verification: An example</td>
<td>179</td>
</tr>
<tr>
<td>5.8</td>
<td>Related work</td>
<td>180</td>
</tr>
<tr>
<td>5.9</td>
<td>Concluding remarks</td>
<td>182</td>
</tr>
<tr>
<td>5.10</td>
<td>Proofs of Chapter 5</td>
<td>184</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Proof of Theorem 5.3 [(x \triangleright m) is decidable]</td>
<td>185</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Proof of Theorem 5.4 [(\forall \psi \Sigma_0 \triangleright^n \Sigma_2 \Rightarrow \Sigma_0 \triangleright^n \Sigma_2 \psi)]</td>
<td>189</td>
</tr>
</tbody>
</table>
The analysis of the interaction of components in open environments can not ignore security issues. Security protocols (are supposed to) enforce desired security properties in charge of protecting the communications of components. They rely on a cryptographic system, and are executed by components which are “connected” together by means of an appropriate sharing of cryptographic keys. Even under the assumption that the underlying cryptographic system is safe, i.e. secret keys can not be guessed, a mis-design of the protocol can expose the interacting components to security failure, like, for example, disclosure and unauthorised manipulation of reserved information, or impersonation of reserved identities. This is mainly due to the fact that components interact within an open, untrustable environment, where potentially all communications can be “intercepted” by other (malicious) components.

In this context, we present a methodology to verify properties of security protocols based on the model of interaction developed in the previous chapters. The expressiveness of the name-based calculus we adopted permits us to adapt the framework with minimal effort to a context where components no longer share references to resources but references to the capability of accessing information, i.e. cryptography keys.

After recalling in more detail security protocols and the related problematic issues, we introduce the cIP-calculus (cryptographic IP-calculus), showing how the IP-calculus can be easily and suitably recasted in order to model key sharing and security protocols. Then, based on a well-known and accepted model of the untrusted environments, we adapt the semantics of component interaction so as to model the presence of an intruder threatening the correctness of the protocol. The framework aims to verify the behaviour of the protocol in any possible environment fulfilling the general and realistic assumptions made.

A precise characterization of the security properties, which a protocol is supposed to guarantee, is still not a completely solved problem. In order to address this point, we introduce a logic predicking over the states of the execution of a protocol. The logic results to be expressive enough in order to characterise many of the most important properties in literature.

Similarly with the verification of behavioural properties discussed in Chapter 3, security protocol verification is based on the model checking of the space of the possible computations. In particular, the validity of a logic formula, which represents the desired security property, is checked in some of the significant states of the execution of the protocol. The execution of the protocol is modeled within an untrusted environment, where the intruder tries to drive the computation towards a state in which the formula does not hold.

Two well-known sources of incompleteness for the verification of security must be mentioned, together with the solutions adopted to deal with them.

Many attacks are based on the possibility of intermixing several concurrent sessions of a protocol, and in general it is not possible to limit the analysis to a given
number of executions. Even if the maximum number of sessions analysed is inevitably a parameter of our framework, the logic has been designed so as to permit the security property to be formalised abstracting from the number of sessions against which they are checked.

The malicious intruder tries to attack the protocol by participating in the communications. In general, the intruder can generate an infinity of different messages, giving rise to an infinite number of different executions, which make their model checking unfeasible. To address this problem, we introduce symbolic analysis techniques: the infinite possibilities are partitioned into a finite set of symbolic, more abstract, executions, which preserve all the information of the non symbolic (concrete) executions. Because of its finiteness, symbolic verification results in being effective (and coherent with respect to the concrete one).

The verification of security properties presented here and that of behavioural properties discussed in Chapter 3 are based on the same principles. Behavioural verification aims to check the correctness, as successful termination of the declared interaction patterns, of a composition of components in an open environment. Composition consists in the sharing of common resources.

Security verification aims to guarantee that the execution of the protocol enforces a given security property, in any open environment where it may be executed by components. Components are connected together by means of a sharing of cryptographic keys.

Differently from behavioural verification, that can be dynamically performed in order to preserve the correctness of an evolving session where new components may or may not be admitted, the interest of security verification is to statically certify a protocol once and forever against all the possible environments (and hence, thanks to the generality of the model adopted, against all the possible intruders).

Some recent results of our research, which deserve further investigation, suggest how this static verification can be used to determine constraints over the sharing of keys, which enforce the desired security property and can be efficiently checked when components dynamically join the system.

The study of formal security protocols verification here presented has been inspired by the problem of properties verification in open systems. Exploiting the features of a name-based formal setting, it has been shown how the same framework can be applied to both open systems and security protocols. The proposed methodology for security protocols verification has been introduced in [BBFT01], where it is also suggested how to generalise the methodology to other kinds of compositional properties. In [BBFT02], the analogy between security protocols and open systems has been further investigated, studying how security and more general properties can be enforced in terms of compositional properties.

This chapter is structured as follows. Cryptography, security protocols and the properties they are supposed to enforce are introduced in Section 5.1, where the
needs for security protocols, and the problems and approaches for their verification are recapped. The definition of the so-called intruder model, i.e. the hypothesis under which a protocol can be attacked in an open environment, is given in Section 5.2.

Section 5.3 presents the cIP-calculus, an extension of the IP-calculus for formally representing security protocols. The calculus extends the IP-calculus with cryptographic primitives in the style of spi-calculus [AG99], and consists of finite components interacting via possibly encrypted communications, with symmetric and asymmetric keys, in an open environment which may be joined by malicious intruders. Since many of the known attacks to the safeness of protocols are carried out by an intruder that mixes-up the communications of different sessions, the calculus provides abstractions to describe multi-session cryptographic protocols.

Section 5.4 presents security properties and the logic used to express them. In spite of its simplicity, the logic permits non-trivial properties on the exchanged data in a protocol to be precisely formalised. The idea of specifying security properties in terms of expected values of exchanged data generalises the approach in [AG99] to a broader set of properties and to multi-session runs of the protocol, where also asymmetric keys are considered.

Then, the verification methodology is presented in Section 5.5. Given the cIP formalisation of the protocol and the formula representing the property, the space of the possible traces of the protocol is checked for the existence of a trace in which the intruder is able to drive the protocol into a state where the property does not hold. Such a methodology facilitates the reasoning about properties by means of the abstract and declarative methods of logic, delegating the procedural verification to an automatic process. An example, shown in Section 5.6, details the phases of the application of the methodology. Unfortunately, as already explained, the procedure — as is — is not effective, due to an infinite branching problem causing an uncontrollable state explosion. Consequently, the example works under the supervision of an oracle, driving the search towards the correct trace.

To overcome the infinite branching problem, techniques of symbolic analysis have been devised in Section 5.7. The symbolic analysis yields a finite-state faithful approximation of the actual state space, where verification can be finitely carried out. After that the theory has been introduced, two examples illustrate its application.

Related work is discussed throughout the paper and in Section 5.8. Section 5.9 sums up the results obtained, and discusses some future developments. Finally, proofs, generally sketched throughout the presentation, are collected in Section 5.10.
5.1 Cryptography + Security protocols = Security?

Interacting in open environments requires protecting communications. As witnessed by the growing use of Internet as primary environment for developing, distributing and running programs, components running in open environments can not make assumptions about the behaviour of the other (possibly malicious) components which populate the same environment.

Communication channels of distributed applications cross the boundaries of many different domains so as to result in practice as being not trustable. All the communications happening through untrusted channels can be “observed” and, if not properly protected, manipulated by “others”, which may impersonate the sender of a message, may change the receiver by redirecting the message, or also may read, substitute or modify the message itself. Network applications, hence, increased the need for robust techniques to state and certify the security of communications.

The way to protect communication relies on the use of cryptography, by means of which data are hidden to those who do not have rights on a given communication. Such rights correspond to the possession of an appropriate cryptographic key which permits the protected messages to be “opened”, read or modified (changing, for example, the sender that is encoded in the message itself). The generation of appropriate keys, typically chosen according to a given probability distribution, is in general an expensive process, so that keys themselves, and their sharing must be accurately managed, [Sti95].

A cryptographic system alone, if not properly used, is not capable of guaranteeing the desired level of security. For instance, a key not properly distributed may spoil the secrecy of a message encrypted by using that key. Moreover, due to the cost of the keys, which depends on the computational difficulties of dealing with big prime numbers, strategies to minimising their number and optimising their usage are needed.

For all this reasons, in the last years, considerable attention has been paid to security protocols, devised in order to guarantee the proper use of the keys and, consequently, the desired security properties.

The initially crafty process of designing and verifying security protocols has then evolved into a process based on a formal methodology, where security properties are formally described and verified against the representation of a protocol.

Several approaches have been developed with the aim of providing semantically well-founded methodologies for verifying security properties. The definition of security models is complicated by the multitude of facets which have an impact on security, and the topic constitutes an open problem. Our work lies in this line of research.
5.1.1 Cryptography

Cryptography takes care of hiding information by encoding the content of the message in a “secret” format and, from centuries, has been extensively investigated and practically applied. A complete introduction to the subject can be found in [MvOV97, Sti95].

A public message $m$ is called cleartext. The process of encoding the message is called encryption. The encrypted message is called cryptogram. Decoding the cryptogram, obtaining back the cleartext, is called decryption. Encryption and decryption usually make use of a key, $k$ say, in that case $\{m\}_k$ indicates the cryptogram of $m$ encrypted with $k$.

In the early stages of use of cryptography, dated back many centuries, communication happened usually among few parts, which shared, beyond the key, the secret of the encoding and decoding algorithms. In modern cryptography, where many different parts may be interested in participating to secure communications, the encoding and decoding algorithms are public and the capability of encrypting and, more importantly, of decrypting depend on the knowledge of the proper key [Ker83]. Key-based encryption algorithms belong to two main classes: symmetric (or secret-key) and asymmetric (or public-key) algorithms.

Symmetric algorithms use the same key for encryption and decryption. The key must hence be known by both the parts participating in a communication. If more parts want to keep private their communications, a key is necessary for each couple, and the number of keys grows with the square of the number of parts.

Asymmetric algorithms use different keys for encryption and decryption, and the decryption key cannot be derived from the encryption key. Asymmetric algorithms (also called public-key algorithms or generally public-key cryptography) permit the encryption key to be public, allowing anyone to encrypt a message with the public key of the receiver, whereas only the proper recipient (who knows the secret decryption key) can decrypt the message. The encryption key is also called the public key and the decryption key the private key. The number of necessary (couples of) keys grows linearly with the number of parts. Asymmetric algorithms are also used to generate digital signatures or certificates: a message encrypted with a private key, $\{m\}_{A^-}$. Everyone, by successfully decrypting the cryptogram with the public key $A^+$, can verify that the message was encrypted by the (only) owner of the private key $A^-$.

Generally, symmetric algorithms are much faster than asymmetric ones. In practice they are often used together, so that a public-key algorithm is used to encrypt a randomly generated encryption key, and the random key is used to encrypt the actual message using a symmetric algorithm. This is sometimes called hybrid encryption. We cite DES (Data Encryption Standard) [Nat77] and RSA [RSA78] among the most well known symmetric and asymmetric algorithms respectively.

In the following we will indicate with $k, k_1, \ldots$ symmetric keys, with $A^-$ the private key of a component named $A$, and with $A^+$ its public key, so that $\{m\}_{A^+}$ is
a cryptogram that only $A$ can decrypt with its private key $A^-$.

Sometimes, encrypted messages are “marked” with freshly generated names. This can be useful for many reasons, like, for instance, time-stamping the message, so as to make it possible to discard old messages (against the risk that an intruder may exploit messages of previous sessions), or for identification purposes, as a sort of unique identifier in a message which can be recognised or can identify the creator of the message. Such names, freshly generated inside a session and used only inside that session, are called *nonces* (from “name once”), and are indicated with $no, na, nb, n_1, n_2, \ldots$.

Moreover, we will assume that the cryptography is “properly working”, i.e. no attack can succeed because of a flaw in the cryptographic system or because something can be deduced from a cryptogram (this, for example, implies that encrypting twice the same message yields two different cryptograms, but these issues are here out of scope), and also, according to literature, that a cleartext can be recognised as such and keys are not structured data, but simply names.

The problem of how keys can be securely distributed is out of the scope of this dissertation (except when done by means of a protocol).

### 5.1.2 Security protocols and their verification

A security protocol consists of a set of participating entities, traditionally called *principals*, a sequence of communications among them, and a property that is supposed to hold as a consequence of the protocol.

Principals behave following a finite and deterministic role. Principal instances interact together, possibly interleaving more instances of the same protocol, by appropriately sharing their cryptographic keys. Security protocols are usually specified in an informal language which describes communications happening among the principals of the protocol. For example, a statement like

$$A \rightarrow B: \{n\}_k$$

intuitively means that a principal $A$ sends to a principal $B$ the datum $n$ encrypted by the key $k$. More precisely, the sentence is actually intended to mean “in *any possible session* of the protocol, an instance of principal $A$ (i.e., a principal behaving according to the specification of $A$) *intends to send* the datum $n$ encrypted with $k$ to an instance of principal $B$”. The underlying hypothesis that communication media are not private channels does not allow a principal, say $B$, to be sure that the message that he receives was sent by $A$. Malicious principals, which do not behave according to the protocol, are usually called *intruders*.

We will consider finite (in the number of their steps) and deterministic (in the behaviour of principals) protocols where *symmetric* and *asymmetric* cryptosystem may be used. As usual, the perfect encryption hypothesis is assumed: *Keys cannot be guessed or deduced* (even if a large number of messages is collected), and a cryptogram can be decrypted only with the proper key.
There is a vast literature showing the many possible aspects which make possible a protocol attack, see [CJ97] for a survey. To have a grasp of the subtleties which must be considered in the verification of security protocols, the next example presents a protocol in the informal language and a possible attack. The example also calls for the need of a proper formalisation of the informal language, which leaves essential details not specified.

Example 5.1 The Denning Sacco Key Distribution [CJ97] protocol is intended to provide a means for sharing a key $k$, generated by $A$ and passed to $B$. A trusted third part $S$ provides certificates for $A$ and $B$. We show a simplified version.

(1) $A \rightarrow S$: $A, B$
(2) $S \rightarrow A$: \{A, $A^+$\}_S-, \{B, $B^+$\}_S-
(3) $A \rightarrow B$: \{A, $A^+$\}_S-, \{B, $B^+$\}_S-, \{\{k\}_A^-\}_B^+$

$A$ says to the trusted server $S$ “I am $A$ and I want to speak to $B$”, (1). $S$ replies saying “I certify, by encrypting with my private key so that everybody can be sure I did it, that $A^+$ is the public key of $A$ and $B^+$ is the public key of $B$”, (2). Finally, $A$ sends $B$ both the certificates and a message encrypted with $B^+$, which only $B$ can decrypt, containing $\{k\}_A^-$, (3). $B$, by decrypting it with the public key $A^+$, gets a key which has been (apparently !) sent by $A$, and is intended to protect the communication among $A$ and $B$.

This simple protocol involves, implicitly, several issues, like, the secrecy of the key $k$, which must kept “restricted” among $A$ and $B$, but also the identities of $A$ and $B$, which are certified by the trusted $S$. Identities also play a part in the protocol, when, for example, by decrypting $\{k\}_A^-$ with the public key $A^+$, $B$ trusts that the message comes from $A$ (from which it has actually been originated).

An attack can be done by a malicious component which plays different parts in subsequent sessions of the protocol. In fact, $B$ can start a new session pretending to be $A$, (we indicate with $B(A)$ a principal (with the knowledge of) $B$ which pretends to be $A$):

(4) $B(A) \rightarrow S$: $A, C$
(5) $S \rightarrow A$: \{A, $A^+$\}_S-, \{C, $C^+$\}_S-
(6) $B(A) \rightarrow C$: \{A, $A^+$\}_S-, \{C, $C^+$\}_S-, \{\{k\}_A^-\}_C^+

$B$ asks $S$ to certify a communication among itself as $A$ and $C$. Intercepting the message sent by $S$ to $A$, and using the previously acquired cryptogram $\{k\}_A^-$, $B$ convinces $C$ of being $A$ and they share the secret key $k$.

In the last years, many techniques for finding flaws of security protocols have been developed [MMS97, Aba99, AG99, AP99, ALV01, BP98, BDNN00, Bor01, MSS98, CJM98, CDL+00, DM99, Hui99, FGM00, Low95, Mea96, RS99, Sch98, Vol99, VS00]. Most of these techniques are based on the analysis of finite state systems, and typically can ensure error freedom only for a finite amount of the behaviour of
protocols. However, an unbound number of principals may take part into interleaved sessions of the protocol, hence the finite state analysis must be able to suitably describing the interleaving of different sessions (we refer to [Low95, Mil99] for a detailed discussion about the relevance of multi-session attacks in protocol analysis).

Some verification techniques are based on the idea of determining the structure of the intruder breaking the protocol. In other words, a security protocol is verified against the presence of an intruder, which, intercepting potentially all the communications exchanged by principals in an untrusted environment, violates a security property of the protocol [DY83].

We will specify security properties by a suitable logical framework. Correctness is therefore reduced to a logical property which is required to hold at the termination of the protocol. The idea of checking for the property only at the termination of the protocol, shared by the majority of the approaches in literature, [Hui99, CJM98], is motivated by the consideration that protocol unsafeness is in general monotone, and if a protocol violates a security requirement during its execution, then such requirement will not hold at the end of the protocol. Moreover, we also consider only the successful terminating states, since we are interested in detecting intruders capable of stealing secrets without being revealed by protocol failures.

Most of the verification techniques work by a state space exploration (i.e. by enumerating all the reachable states of a system). Hence, a main problem is the so called state explosion problem: the state space may grow very rapidly. We summarize below which are the, generally accepted, potential causes of infiniteness (e.g. [Hui99]), discussing how our proposal handles these issues.

• **Computational Power of the Principals**: In general, a principal could be a Turing complete computational entity. We have instead limited its computational power by avoiding recursive constructs. This in particular implies that the execution of a protocol cannot diverge. This is not a severe limitation because protocols are in general finite fragments of interactions where different principals agree on certain premises, like for example a shared session key.

• **Number of Principals**: The number of principals in a protocol is in general finite, nonetheless many known attacks are based on exploiting the mixed interleaving of different sessions of the same protocol. On one hand, there are approaches that introduce restrictive hypotheses on the behaviour of the principals allowing one to generalise a safety result for a limited number of sessions to the unbound case, [Low98]. On the other hand, there are results that show that in the general case, [Low95, Mil99], it is not possible to deduce the safety of the protocol from the safety of a bounded number of interleaved sessions, and that some attacks are possible only thanks to the parallel execution of more session of the same protocol [Mil99].

Our approach focuses on termination, hence we treat the maximum number of parallel session for which we want to certify protocol security as a parameter
of the methodology. Among the methods that give up termination in favour of completeness in the analysis of multi-session protocols we cite [Mea96, Low96]

- **Number of Possible Messages:** Whenever an intruder sends a message to another principal, it can derive from its knowledge an arbitrarily complex message, giving rise to an infinite number of choices. In our approach communications are “typed” by the received message, i.e. only a message matching the receiving message can be communicated. Nonetheless this is not sufficient to bound the number of possible choices. In particular, when the receiver waits for an unspecified message, any message can in principle be sent. To overcome this problem we have adopted a symbolic analysis: When an intruder message must match a not specified message, then an abstract representation of the current knowledge of the intruder will be sent. Every actual message is derivable from this abstract representation.

Moreover, the generative power of the intruder can also be enhanced by providing the intruder with a knowledge containing a given number of fresh names it can use.

### 5.1.3 Security properties

Formalising the security properties that protocols should enforce is even more difficult than formalising the protocols themselves. Many properties have a clear intuitive meaning that often is difficult to be expressed in terms of behavioural concerns only. Even if there is a widespread understanding of the informal meaning of properties, their proposed formalisations vary a lot. We concentrate on **secrecy, integrity** and **authentication**, as some of the more general properties, of which we give an informal explanation.

- Secrecy regards the sensible data exchanged in the protocol. Secrecy of a message, typically an encrypted message \( \{m\}_k \), reads as the impossibility for an intruder of deducing any information about \( m \) after that the protocol has been executed. For example, in the following protocol, secrecy of \( m \) can not be guaranteed, since at the end of the protocol both \( \{m\} \) and \( k \) may have been observed by the intruder, which can hence derive \( m \).

\[
\begin{align*}
(1) & \quad A \rightarrow B : \quad \{m\}_k \\
(2) & \quad A \rightarrow B : \quad k
\end{align*}
\]

Secrecy is supposed to guarantee not only that \( m \) can not be known, but also that no information leaks from the protocol. Sometime this may also depend on the correctness of the cryptography system. For example, if the same message \( \{m\}_k \) is sent twice during a protocol, an intruder can not deduce that the two messages are indeed the same (and hence encrypting twice the same message with the same key should yield two different cryptograms).
5.1. CRYPTOGRAPHY + SECURITY PROTOCOLS = SECURITY?

- Integrity regards the impossibility for an intruder of manipulating the messages, typically the encrypted ones, which are exchanged in the protocol. For example, in the following protocol

\[
\begin{align*}
&\text{(1)} \quad A \rightarrow B : \{m\}_k \\
&\text{(2)} \quad A \rightarrow C : \{n\}_k
\end{align*}
\]

integrity of the data received by $B$ $(C)$ can not be guaranteed. In fact, even without knowing $k$, an intruder could swap the messages, forwarding to $B$ data sent for $C$, and vice-versa, without that the two may be aware of that. It is often possible that an intruder can destroy messages, preventing the receivers to get them, but this does not appear as a real attack (supposed that the intruder does not gain knowledge from doing that), but rather a protocol failure. The possibility that an intruder spoils integrity by re-using messages of previous sessions of the protocol is typically dealt with the use of nonces.

- Authentication reads as the capability of correctly identifying the sender of a messages received by a principal. Example 5.1 showed an authentication protocol failure.

Authentication results in general difficult to be formalised and guaranteed, and many known attacks are based on the impersonation by the intruder of the identity of a legal participant in the protocol, or on the intruder playing the part of “man in the middle”, who exploits the knowledge acquired in a session of the protocol, to drive another session towards a successful attack, like in the example.

In general, simply checking the content of received communications may not be enough for authentication, which may depend on complex relations among data and principals of the protocol.

Without attempting at a more formal characterisation of properties, we will present a logic, predicating over both the principals, their shared keys and the exchanged messages, as a general means to formalise the desired properties, leaving to the protocol analyser the liberty and the responsibility of finding the proper characterisation of the property of interest, being then the rest of the verification process fully automated.

5.1.4 Discussion: (c)IP-calculus and security

The application of the IP-calculus to security protocols aims both to illustrate a practical application of the framework, showing how its features can be used in a case of formal verification for open systems, and to contribute to the foundations of the formal verification of security protocols in the more general perspective of component-based systems. Static protocol verification consists of three steps
1. Specification of a protocol \( \mathcal{P} \),

2. specification of a property \( \phi \) that \( \mathcal{P} \) should satisfy, and

3. verification of whether \( \mathcal{P} \) does (or does not) satisfy \( \phi \).

Even if the steps differ in some aspect from the, possibly dynamic, verification of behavioural properties illustrated in Chapter 3, nonetheless, the IP-calculus based framework suitably applies to the formal verification of security protocols. Indeed,

- security protocols consist of a finite sequence of communications performed by the principals of the protocol, that may hence be naturally described as interaction patterns. The IP-calculus can be straightforwardly extended in order to deal with cryptographic primitives, while the presence of untrusted channels simply corresponds to the existence of global names in the name space of every principal (interaction pattern),

- sharing of keys is a sharing of names. All the mechanisms for the sharing and communication of channel names apply to the sharing and communications of cryptographic keys. For example, two principals share the same key by mapping their open (key) variables to the shared key (in the same way in which two interaction patterns share the same channel by connecting together their open variables),

- the composition of the principals and their interaction in executing the protocols correspond to the construction of a session, exactly as for the interaction patterns. The session, which evolves according to the protocol, is joined by the patterns of the principals and their keys are shared.

Moreover, as illustrated in Section 5.9, we are currently investigating how protocol verification can pave the way for a dynamic style of component verification in charge of enforcing security properties.

### 5.2 The Intruder model

Following the Dolev-Yao model [DY83], the environment is the intruder, in its capability of intercepting (virtually all) the principal communications. Consequently, the intruder is characterised by the knowledge it acquires by intercepting the messages sent by the principals, and that it may use in order to successfully attack the protocol (see, for instance, [Low95] for the description of a famous attack). More precisely, the intruder/environment can

- receive, destroy and re-transmit messages circulating over the net, and

- remember and decompose acquired messages, and create new messages from the acquired ones.
While the former actions will be modeled by an appropriate definition of the semantics, making explicit the communication capability of the intruder, the latter are modeled by the representation of the intruder knowledge and the operations of updating the knowledge with a received message and of deriving a message from a given knowledge. Such knowledge, throughout the chapter, is represented by the set (of messages) $\kappa$.

Moreover, the set $\kappa$ can also be used to provide the intruder with a generative capability, by adding to the knowledge a set of fresh names or keys that the intruder can generate and use in deriving messages. Actually, the problem of bounding the number of new names needed in order to successfully performing an attack is still an open problem, and the solution we adopted is the one chosen in the vast majority of the proposals in literature, e.g. [BB02].

For the purpose of verification, the initial $\kappa$ provided to the intruder, and often reffered to as $\kappa_0$, can also be used to test the robustness of a protocol with respect to different classes of attacks based on the power of the intruder, 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.

Coherently with most of the literature, we also assume that any attack can be performed by a single intruder. Intuitively, augmenting the number of intruders does not result in acquiring more knowledge than that acquired by a single intruder interfering with all the communications happened in the protocol. As a strong limit to the power of the intruder, we assume the perfect encryption hypothesis: Keys cannot be guessed or deduced (even if a large number of messages is collected).

First, we introduce the set of messages circulating in a protocol and formally define the set $\kappa$, and the updating of $\kappa$ with a message $m$.

**Definition 5.1 (Messages)** The set of ground messages $M$ is defined according to the following syntax:

$$M ::= PN | K | PN^+ | PN^- | NO | \{M\}_K | \{M\}_{PN^+} | \{M\}_{PN^-} | (M,M)$$

where $PN,K,PN^+,PN^-,$ NO represent principal names, symmetric keys, public and private keys, and nonces, while $\lambda_-$ and $\lambda_+$ are the encrypting and pairing operators, respectively. The metavariable $\lambda$ ranges over $K \cup PN^+ \cup PN^-$ and $\lambda^- = \lambda$ if $\lambda = k \in K$, $\lambda^- = A^{-}$ if $\lambda = A^+ \in PN^+$ and $\lambda^- = A^{-}$ if $\lambda = A^{-} \in PN^-$. The metavariables $m,n,p,q,o$ range over messages.

Principal names consist of infinitely many constants $A, B, S \ldots$ or $I$, the name for the intruder. Note that principal names, opportunely decorated with $+$ or $-$, represent the public and private keys of the principal itself (in general we assume that the knowledge of a principal name implies the knowledge of its public key, and that only a principal knows its private key, unless it deliberately communicates it).

**Definition 5.2 (Knowledge)** A finite set $\kappa \subseteq M$ is called (intruder) knowledge. Given $m \in M$, the knowledge $\kappa' = \kappa \cup m$ is the update of $\kappa$ with $m$. 
The concept of derivation of a message from a knowledge, $\kappa \triangleright m$, straightforwardly follows from the data constructors and destructor used in the syntax of the messages. Trivially, a message can be derived from $\kappa$ if it belongs to $\kappa$. If two messages can be derived, then also the couple containing them can be derived. A cryptogram can be derived if its content and key can be derived.

On the other hand, messages can be derived also by de-structuring complex messages. Thus, from a tuple belonging to $\kappa$ it is possible to derive the messages it contains and the content of a cryptogram can be derived if the cryptogram and the corresponding decrypting key can be derived. In a word, the notion of message derivation of the following Definition 5.3 models the set of data that an intruder with knowledge $\kappa$ can forge.

**Definition 5.3 (Message derivation ($\triangleright$))** A message $m$ can be derived from a knowledge $\kappa$ if and only if $\kappa \triangleright m$ can be proved by the following rules

\[
\frac{m \in \kappa}{\kappa \triangleright m} \quad \frac{\kappa \triangleright m \quad \kappa \triangleright n}{\kappa \triangleright (m,n)} \quad \frac{\kappa \triangleright \lambda}{\kappa \triangleright \{m\}_\lambda} \quad \frac{\kappa \triangleright \lambda}{\kappa \triangleright \{\lambda\}}
\]

A message $m$ can be constructively derived from a knowledge $\kappa$, $\kappa \triangleright_i m$, if and only if $\kappa \triangleright m$ can be proved by means of the constructor introduction rules only, namely $\in$, $( )_i$ and $\{\}$.i.

Constructive derivations are important because they are decidable and their decidability is used to prove the decidability of $\triangleright$.

**Theorem 5.1 ($\triangleright_i$ is decidable)** Given a knowledge $\kappa$ and a message $m \in M$, $\kappa \triangleright_i m$ is decidable.

**Proof.** The constructive derivation of a message is monotone (and $\kappa$ is a finite set).

In [CJM98], in a model which does not take into account asymmetric keys, it has been shown that the derivation of a message from a knowledge is decidable. That proof relies on the similarity of message derivation with natural deduction, and in fact it is easy to verify that, in absence of asymmetric keys, every constructor has an introduction and a corresponding elimination rule, in the style of natural deduction.

For example, given the cryptogram $\{m\}_k$ and the key $k$, rule $\{\}_e$ derives the message $m$, and correspondingly, from the key $k$ and the (derived) message $m$, rule $\{\}_i$, derives the cryptogram $\{m\}_k$. 

Intuitively speaking, this suggests that there is not loss of information in the use of constructor elimination rules, and hence, being reversible operations, they can be reordered in a proof, so that, after a finite number of their applications, the proof proceeds constructively only, resulting then decidable (similarly with what happens for natural deduction, see the above mentioned paper for details).

In presence of asymmetric keys things get more complicated. For example, from the cryptogram \( \{m\}_{A^{-}} \) and the asymmetric (public) key \( A^{+} \), rule \( \{\} \) derives the message \( m \), but from the (derived) message \( m \) and the key \( A^{+} \), neither rule \( \{\} \), nor any other rule, is able to reconstruct the original cryptogram \( \{m\}_{A^{-}} \).

In the following we prove that derivation is decidable in presence of asymmetric keys, extending the result of [CJM98]. The proof is based on a “remembering” transformation of \( \kappa \), the \( e \)-elimination function, which applies the elimination rules, taking care not to forget messages that can not be reconstructed. It will be shown that derivation from \( \kappa \) reduces, in a correct and complete way, to constructive derivation from \( e(\kappa) \), which is decidable.

**Definition 5.4** Given a knowledge \( \kappa \), its explicit form is the knowledge \( \kappa' = e(\kappa) \), where the function \( e(\_\_) \) from a knowledge to a knowledge is defined as follows:

\[
e(\kappa) = \begin{cases} 
\kappa & \text{otherwise} \\
\kappa \setminus m \cup p \cup q \cup m & m = (p, q) \in \kappa \\
\kappa \setminus m \cup n \cup m & m = \{n\}_\lambda \in \kappa \land \lambda^- \in \kappa 
\end{cases}
\]

**Theorem 5.2** \( e(\_\_) \) is well defined

**Proof.** (sketch, see Section 5.10)

The proof consists of two steps:

1. showing that the function always terminates, and
2. showing that, despite its non-deterministic definition, the function always returns an unique value.

\( \square \)

The aim of the transformation of \( \kappa \) in an explicit form is twofold: it helps in proving the decidability of \( \triangleright \) in presence of asymmetric keys and it provides an effective method for implementing message derivation that will be used as part of the semantics of the intruder in our model. Next example shows, on a simple case, the transformation performed by the elimination function.
Example 5.2 Let us consider the set
\[ \kappa = \{ \text{no}_1, kb, \{(\text{no}_2, \text{no}_1)\}_{A^+_2}, \{A^-_2\}_{kb} \} \]
where, the presence of the message \( \{A^-_2\}_{kb} \) and of the key \( kb \) allows for the application of constructor elimination rules. Applying the elimination function, we get \( \kappa' \) the explicit form of \( \kappa \).

\[ \kappa' = e(\kappa) = \]
\[ = e(\{ \text{no}_1, kb, \{(\text{no}_2, \text{no}_1)\}_{A^+_2}, A^-_2 \}) \cup \{ A^-_2 \}_{kb} \]
\[ = e(\{ \text{no}_1, kb, (\text{no}_2, \text{no}_1), A^-_2 \}) \cup \{ (\text{no}_2, \text{no}_1) \}_{A^+_2}, \{ A^-_2 \}_{kb} \]
\[ = e(\{ \text{no}_1, kb, no_2, no_1, A^-_2 \}, (\text{no}_2, \text{no}_1), \{ (\text{no}_2, \text{no}_1) \}_{A^+_2}, \{ A^-_2 \}_{kb} \]
\[ = \{ \text{no}_1, kb, no_2, no_1, A^-_2, (\text{no}_2, \text{no}_1), \{(\text{no}_2, \text{no}_1)\}_{A^+_2}, \{A^-_2\}_{kb} \} \]

It is easy to realise that each message derivable from \( \kappa \) can be constructively derived from \( \kappa' \).

The decidability of \( \triangleright \) is stated by Theorem 5.3. The following two lemmas facilitate its proof. Informally, they respectively state that 1) the construction \( e(\_\_) \) does not forget any message, 2) the construction \( e(\_\_) \) does not introduce the possibility of deriving new messages.

**Lemma 5.1** Let \( m \in M \) be a message, and \( \kappa \) a knowledge, then \( m \in \kappa \Rightarrow m \in e(\kappa) \).

**Proof.** By definition of \( e(\_\_) \) (no element is deleted at any time).

\( \square \)

**Lemma 5.2** Let \( m \in M \) be a message, and \( \kappa \) a knowledge, then \( m \in e(\kappa) \Rightarrow \kappa \triangleright m \).

**Proof.** (see Section 5.10 for the complete proof.)

(by structural induction on \( e \)).

The set \( e(\kappa) \) can have been obtained by applying one of the three rules of the definition of the function \( e(\_\_) \). We present here one of them:

\[ e(\kappa) = e(\kappa \setminus o \cup p \cup q) \cup o \]

The side conditions for this case imply that \( o = (p, q) \in \kappa \). Either \( m = o \in \kappa \), but then \( \kappa \triangleright m \) (by rule \( \epsilon \)), or \( m \in e(\kappa \setminus o \cup p \cup q) \). By the inductive hypothesis, it follows that \( \kappa \setminus o \cup p \cup q \triangleright m \), and equivalently \( \kappa \cup p \cup q \triangleright m \). Note that,
since \( o = (p, q) \in \kappa \), by applying \( (.)_e^1 \) and \( (.)_e^2 \), it also holds \( \kappa \triangleright p \) and \( \kappa \triangleright q \). Trivially (reasoning on the proofs):

\[
\begin{align*}
\kappa & \triangleright p \\
\kappa & \triangleright q \\
\kappa \cup p \cup q & \triangleright m
\end{align*}
\]

\( \Rightarrow \kappa \triangleright m \)

\( \square \)

Finally, decidability of \( \triangleright \) can be proved.

**Theorem 5.3 (\( \kappa \triangleright m \) is decidable)** Let \( m \in M \) be a message, and \( \kappa \) a knowledge, then

\[ \kappa \triangleright m \iff e(\kappa) \triangleright_1 m, \]

(and hence, by Theorem 5.1, \( \kappa \triangleright m \) is decidable).

**Proof.** (see Section 5.10 for the complete proof.)

\[ \kappa \triangleright m \Rightarrow e(\kappa) \triangleright_1 m \]

(by induction on \( i \), the length of the proof for \( \kappa \triangleright m \)).

\( i = 1 \).

The only proof of length one is

\[ \frac{m \in \kappa}{\kappa \triangleright m} \in, \]

and then, by Lemma 5.1, \( m \in e(\kappa) \), and hence \( e(\kappa) \triangleright_1 m \).

\( i \Rightarrow i + 1 \).

Let us suppose that the message \( m \) is derived by a proof of \( i + 1 \) steps, which can terminate by applying one of the remaining five rules. We show here the case for rule \( \{ \} \).

If \( m = n \) is obtained by applying the rule

\[
\frac{\kappa \triangleright \{n\}_\lambda \quad \kappa \triangleright \lambda}{\kappa \triangleright n} \in, \]

then, by inductive hypothesis \( e(\kappa) \triangleright_1 \{n\}_\lambda \) and \( e(\kappa) \triangleright_1 \lambda \). The proof for \( e(\kappa) \triangleright_1 \{n\}_\lambda \) can be concluded in two ways. A case occurs when applying the rule

\[ \frac{e(\kappa) \triangleright_1 n \quad e(\kappa) \triangleright_1 \lambda}{e(\kappa) \triangleright_1 \{n\}_\lambda} \in, \]
which implies that $e(\kappa) \triangleright_i n$. The other case occurs when the rule

$$\frac{\{n\}_\lambda \in e(\kappa)}{e(\kappa) \triangleright_i \{n\}_\lambda}$$

is applied. If $\{n\}_\lambda \in e(\kappa)$, from the definition of $e(\_)$, it follows that also $n \in e(\kappa)$ (if not the recursive construction of $e(\_)$ could not have terminated). It follows that $e(\kappa) \triangleright_i n$.

$$e(\kappa) \triangleright_i m \implies \kappa \triangleright m$$

(by induction on $i$, the length of the proof for $e(\kappa) \triangleright_i m$).

$i = 1$.

The only proof of length one is

$$\frac{m \in e(\kappa)}{e(\kappa) \triangleright_i m} \in,$$

and then, by Lemma 5.2, $\kappa \triangleright m$.

$i \Rightarrow i + 1$.

Two rules can be applied to derive $e(\kappa) \triangleright_i m$ in more than one step, namely $(\_)_i$ and $\{}_i$. We show here the first case $m = (p, q)$:

$$\frac{\kappa \triangleright_i p \quad \kappa \triangleright_i q}{\kappa \triangleright_i (p, q)} (\_)_i,$$

then, by inductive hypothesis $\kappa \triangleright p$ and $\kappa \triangleright q$, and hence

$$\frac{\kappa \triangleright p \quad \kappa \triangleright q}{\kappa \triangleright (p, q)} (\_)_i.$$

Analogously, if $m = \{n\}_\lambda$ and $\{}_i$ is used, then

$$\frac{e(\kappa) \triangleright_i n \quad e(\kappa) \triangleright_i \lambda}{e(\kappa) \triangleright_i \{n\}_\lambda} \{}_i.$$

By inductive hypothesis $\kappa \triangleright n$ and $\kappa \triangleright \lambda$, and hence

$$\frac{\kappa \triangleright n \quad \kappa \triangleright \lambda}{\kappa \triangleright \{n\}_\lambda} (\_)_i.$$
Example 5.3 Following Example 5.2, after having proved the decidability of $\triangleright$, let us see how a message can be derived from a knowledge $\kappa$ and constructively derived from $\kappa' = e(\kappa)$. Let $m = \{no_2\}_{A_2^-}$ and

$$\kappa = \{no_1, kb, ((no_2, no_1))_{A_2^+}, \{A_2^-\}_{kb}\},$$

then the following is a proof for $\kappa \triangleright m$:

$$\frac{\{(no_2, no_1)\}_{A_2^+} \in \kappa}{\kappa \triangleright \{(no_2, no_1)\}_{A_2^+}}$$

$$\frac{\kappa \triangleright \{(no_2, no_1)\}_{A_2^+}}{\kappa \triangleright no_2} \quad (e_1)$$

$$\frac{\kappa \triangleright \{(no_2, no_1)\}_{A_2^+}}{\kappa \triangleright A_2^-} \quad \{e\}$$

On the other hand, being

$$\kappa' = e(\kappa) = \{no_1, kb, no_2, no_1, A_2^-, (no_2, no_1), ((no_2, no_1))_{A_2^+}, \{A_2^-\}_{kb}\},$$

the following is a proof for the constructive derivation $\kappa' \triangleright_i m$:

$$\frac{no_2 \in \kappa'}{\kappa' \triangleright_i \{no_2\}}$$

$$\frac{A_2^- \in \kappa'}{\kappa' \triangleright_i A_2^-} \quad \{i\}$$

It is easy to observe how all the elimination rules have been applied by the elimination function, and hence in the derivation from $\kappa'$ only introduction rules are needed.

5.3 cIP: A Cryptographic calculus of Interacting Principals

The extension of the IP-calculus towards a calculus for security protocols, the cIP-calculus, consists of a simple modification of the syntax, introducing cryptographic operators and mechanisms for the sharing of keys, and in correspondingly adapting the semantics to the new syntax. This extension is essentially again a name-passing process calculus in the style of the $\pi$-calculus [MPW92]. A similar operation has been done in the inspiring [AG99], where the spi-calculus has been defined by extending the $\pi$-calculus with cryptographic primitives. Differently from that work, we did not introduce ad-hoc cryptographic primitives as actions of the principals, since encryption and decryption is embedded are communications, and, more precisely, in the way in which encrypted data can be or can not be received depending on the possession of the corresponding decrypting key. In this way, it is possible to abstract
from the needs of programming the encryption and decryption steps, and to adapt the semantics with minimal modifications.

The execution of a protocol consists of a session that can be joined by more principals. Principals are, actually, instances of cryptographic interaction patterns representing the behaviour of the regular principals, according with the definition of the protocol. Such approach results 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 a multi-session execution. Moreover, the possibility of intruders that do not necessarily act in any prefixed way is taken into account by a suitable semantics, according to the model presented in Section 5.2.

5.3.1 Syntax

In presenting the syntax, we focus on an abstract signature concerning the necessary elements for expressing security protocols. Hence, for instance, the silent action does not occur in the behavioural expressions, and names and variables stand for (and range over) keys only, and not channel names. In particular, channel names are not present even in communication actions, since it is assumed that all the communications happen over a public channel that is known by every principal in the session. Accordingly, the join operation will only share keys. It is understood that all the channel machinery can be reintroduced when needed.

First, we extend the set of messages already introduced with variables, so as to obtain the set of data which may occur in principal bodies. Variables are named according to the principal in which they occur, so that, for instance, \( x_a \) is a variable of principal \( A \), while \( x_b \) belongs to principal \( B \).

Usually, \( \pi \)-calculus like languages do not make any distinction between variables and names. Following [AG99], 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 assume that each principal instance that takes part in a computation uses private variables that may be instantiated to names shared with other participants, keeping the two distinct facilitates the presentation.

Since we are interested in the possibility of modeling multi-session executions of protocols, each instance of a participating principal will be labeled with a unique identifier, namely a natural number. For this reason, the set of data consists of labeled elements, according to the following definition.

**Definition 5.5 (Data)** Let \( V = \{w_a, w_b, x_a, x_b, y_a, y_b, \ldots\} \) be a set of variables, then the set of data \( M_V \) is defined according to the following syntax:

\[
M_V ::= PN \mid K \mid PN^+ \mid PN^- \mid NO \mid V \mid \mathcal{?}V \mid \{M_V\}_{M_V} \mid (M_V, M_V)
\]

where \( PN, K, PN^+, PN^-, NO \) are as in Definition 5.1, and \( \mathcal{?}V \) denotes a binding occurrence of a variable.\(^1\)
5.3. CIP: A CRYPTOGRAPHIC CALCULUS OF INTERACTING PRINCIPALS

\[
E ::= 0 \mid \alpha.E \mid E||E \mid E+E \\
\alpha ::= \text{in}(M_V) \mid \text{out}(M_V)
\]

Figure 5.1: Behavioural expressions of principals in cIP

The unusual names for variables are motivated by the assumption, discussed in the following, that from the name of a variable it is possible to derive the principal in which it occurs, so that, for instance, the variable \(xa\) is a variable occurring in principal \(A\).

Simply, the definition of a principal, i.e. an interaction pattern of a component participating into a cryptographic protocol, is given by the next definition. Note how the behavioural expressions have been simplified according to what explained above.

**Definition 5.6 (Principal)** A principal \(P\) of a security protocol is defined as:

\[
P ::= PN \triangleq (X)[E]
\]

where \(X\) a set of variables called the open variables of the principal, and \(E\) is a behavioural expression as defined in Figure 5.1,

The open variables \(X\) of a principal \(A \triangleq (X)[E]\) explicitly declare the keys the principal \(A\) needs to share with other principals. Principals are connected one to another by assigning names (viz., symmetric keys or principal names to be used as public keys) to their open variables. Conventionally, variables for asymmetric keys are decorated in a behavioural expression with \(-\) or \(+\), depending of their use as a private or public key. We say that \(x \in X\) is a variable for a symmetric key if \(x^+\) and \(x^-\) never occur in \(E\), otherwise we say that \(x\) is a variable for an asymmetric key. We assume that variables for symmetric keys cannot appear as variables for asymmetric keys and vice-versa.

The communication actions \(\text{out}(M_V)\) and \(\text{in}(M_V)\) output the datum \(M_V\) to the environment or receive it from the environment. Since it is not worth distinguishing different public channels, all of which could be accessed by every principal, channel names are omitted in communication actions.

Names (keys) are still used to encode secrets, and may be extruded and change the communication topology. For instance, a principal \(A\) may extrude a private key

---

1Note that, differently from \(M\), \(M_V\) contains also terms like \(\{NO\}_{(M,M)}\) which may be produced by the interaction of principals, for example when one receives a message \(((m,m))\) that it will try to use as key. As it will be shown by semantics, these cases resolves into run-time errors, in the sense that they lead to states from which no transition is possible.
$k$ to a principal $B$ by sending the message $d = \{k\}_{B^+}$. $B$ is the only principal that may decrypt $d$ and then use $k$ as encryption or decryption key (provided that it has not published its private key).

This phenomenon is quite similar to the scope extrusion mechanism of the $\pi$-calculus. The relevant difference is that a private channel cannot be accessed by components that do not possess it, while cryptograms like $d$ may be stolen, even if not understood, by intruders because they are exchanged using public communication channels. The spi-calculus, even if it has channel names, uses restriction essentially for keys or nonces, similarly to what happens in our case where names (keys) are local. In [Bor01] channel names are used to determine the principals that perform communication on the public channel, that, like in our case, is the only one used for exchanging data.

A behavioural expression (Figure 5.1) may consist of the inaction $0$ (sometimes omitted in the behavioural expressions, if not essential) or of a communication action $\alpha$ prefixed (\texttt{"."}) to a behavioural expression, or it may be a parallel ($||$) or non-deterministic choice (+) composition operator.

The notation "$?\tau$" is used to denote a binding occurrence of a variable inside a $\text{in()}$ operation. Variable bindings work as usual, with some particular constraints.

**Definition 5.7 (Bound and free variables)** A binding occurrence of a variable $x$ is an occurrence of $?x$ in an input action $\text{in}(d)$. A binding occurrence in an input action extends its scope to all the actions to which the input action is prefixed and to all the (non-binding) occurrences of the variable in the same action (no more than a binding occurrence can appear into the same action).

An occurrence of a variable $x$ is bound in a principal $(X)[E]$ if it occurs in the scope of a binding occurrence $?x$ or if $x$ occurs in $X$. It is free otherwise. The set $\text{bv}((X)[E])$ is the set of the variables with bound occurrences in $(X)[E]$, and $\text{fv}((X)[E])$ is the set of variables with free occurrences in $(X)[E]$.

A principal is closed if it does not contain free (occurrences of) variables.

As usual, two principals are structurally equivalent if one is obtained by the other by alpha-renaming the bound variables.

For example, in the expression $\text{in}(?y).\text{in}(y).0$ the second occurrence of $y$ is bound by the first occurrence of $y$. This means that the second action will attempt to receive the same datum received by the first one (and assigned to $y$). Differently, the expression $\text{in}(?y).\text{in}(?y).0$ corresponds to receive two possibly different data, one for each binding occurrence of the variable $y$.

Throughout this chapter we will always assume that:

- no input action can contain two binding occurrences of the same variable, in order to avoid unnecessarily complex scoping rules (note that they could anyway be renamed),
• output actions do not contain binding occurrences, coherently with the adopted use of bindings for instantiating “received” values,

• variables occurring as encryption keys in input actions cannot be binding occurrences, because, according with the embedding of cryptographic primitives into matching communications, a principal is required to explicitly declare, via a ground message, the key it intends to use for decrypting a cryptogram,

• from the name of a variable it is possible to deduce the principal in which it occurs (see Definition 5.5),

• principals are closed.

Substitution, sometimes written \([x \rightarrow v]\), works as expected, \(\epsilon\) is the empty substitution.

**Example 5.4** Let us consider a simple protocol and its formalisation:

\[
\begin{align*}
(1) & \quad A \rightarrow B : \{m\}_k \\
(2) & \quad B \rightarrow A : \{n\}_{A^+}
\end{align*}
\]

\[
A \triangleq (xa)[out(\{m\}_xa).in(\{?ya\}_{A^-})] \\
B \triangleq (zb,wb)[in(\{?vb\}_{zb}).out(\{n\}_{wb^+})]
\]

A sends B a nonce \(m\) encrypted with the symmetric key \(k\). Then B sends A the nonce \(n\) encrypted with the public key of A. In the formalisation, principals A and B have the open variables \(xa\) and \(zb\), respectively, to share the symmetric key \(k\). The open variable for asymmetric key \(wb\) of B will be instantiated with the name (hence the public key) of A.

### 5.3.2 Sessions and Multiple-sessions

Principals interact by joining a session populated by possible partners. More instances of the same principal may non-deterministically access the same session, so that more instances of the protocol can be concurrently running in what is commonly called a multi-session execution of the protocol.

In order to model multi-sessions it is necessary to define an appropriate notion of principal instance and a mechanism implementing the connection of a new principal instance to a (running) multi-session.

The latter is, as usual, modeled by means of a join operation, that takes care of connecting open variables of (renamed apart) principal instances. The former consists in indexing the data of each instance with a unique natural number, so that local names and variables (of different instances of the same principal) are syntactically distinguished. This mechanism guarantees also that instances are renamed apart. The first step for defining a protocol instance is to define indexed data.
Definition 5.8 (Indexed Variables, Messages and Data)  The set of indexed variables $V_i$ is defined as follows

$$V_i = \{xa_i | xa \in V \land i \in \mathbb{N}\}.$$  

The set of indexed messages $M_i$ is defined as follows

$$M_i ::= PN_i \mid K \mid PN_i^+ \mid PN_i^- \mid NO_i \mid \{M_i\}_K \mid \{M_i\}_{PN_i^+} \mid \{M_i\}_{PN_i^-} \mid (M_i, M_i).$$

The set of indexed data $M_{Vi}$ is defined as follows

$$M_{Vi} ::= PN_i \mid K \mid PN_i^+ \mid PN_i^- \mid NO_i \mid V_i \mid ?V_i \mid \{M\}_M \mid (M, M)$$

where $PN_i, PN_i^+, PN_i^-, NO_i, V_i$ represent principal names, public and private keys, nonces and variables indexed with a natural number.

Note that symmetric keys, which are not associated with an instance (non-local names), are not indexed.

Definition 5.9 (Instances and Sessions) Given a principal $A \triangleq (X)[E]$ and a natural number $i$, an instance $A_i$ of $A$ is $(X_i)[E_i]$, where all variables and data occurring in $X$ and $E$, are indexed by $i$.

A session is a (possibly empty) set $\{(X_1)[E_1], ..., (X_n)[E_n]\}$ of instances. If two instances are indexed with different indexes, say $i \neq j$, then all the indexed data of $(X_i)[E_i]$ are distinguished from those of $(X_j)[E_j]$, with the exception of the symmetric keys which they possibly share.

The join operation defines how a principal instance can enter a (running) session by connecting open variables for asymmetric keys to principal names and open variables for symmetric keys to keys in $K$. Connected variables are not open anymore.

Definition 5.10 (Join) Let $\mathcal{C} = \{(X_{i_1})[E_{i_1}], ..., (X_{i_{n-1}})[E_{i_{n-1}}]\}$ be a session, let be $n = \max j_i j + 1$, and $A_n \triangleq (Y_n)[F_n]$ a principal instance and

$$\gamma : \bigcup_{i=1}^{n-1} X_i \cup Y_n \rightarrow (PN_i \cup K)$$

a partial mapping (not necessarily defined over all the open variables) such that, for all $xa \in \text{dom}(\gamma)$, $\gamma(xa) \in K$ if $xa$ is a variable for a symmetric key and $\gamma(xa) \in PN_i$, otherwise. Then:

$$\text{join}(A_n, \gamma, \mathcal{C}) = \bigcup_{i=1}^{n-1} \{(X_i - \text{dom}(\gamma))[E_i \gamma]\} \cup \{(Y_n - \text{dom}(\gamma))[F_n \gamma]\}$$
A common assumption over public key cryptosystems is that principals do not initially know the private key of any other principal. A principal \(A\) can learn the private key \(B^-\) of another principal \(B\), only if \(B\) will send such key to \(A\). Therefore, in general, an open variable should never be instantiated with, or used as, a private key unless, analogously with what can be done with the initial intruder knowledge (see Section 5.2), one wishes to study the protocol under the assumption that a private key of a principal is known by another principal.

The next example shows how more sessions of a protocol can be combined by joining together more instances of the principals the protocol itself.

**Example 5.5** Referring to Example 5.4, we show how to join an instance of \(B\) into the (running) session \(C = \{A_1, B_2, A_3\}\), where the instance \(A_1\) shares with the instance \(B_2\) the key \(ka\), by means of which it sends the nonce \(m_1\). \(B_2\), in turn, intends to send the nonce \(n_2\) to \(A_3\). The only open variable in the session is \(x_3\) of \(A_3\):

\[
\{ (\text{out}(\{m_1\}_{ka}).\text{in}(\{?ya_1\}_{A^-_1}))], (\text{in}(\{?vb_2\}_{ka}).\text{out}(\{n_2\}_{A^+_3}))],
(xa_3)\text{out}(\{m_3\}_{xa_3}).\text{in}(\{?ya_3\}_{A^-_3})\}
\]

Instance \(B_4\) is joined to the session by instantiating the open variables \(xa_3\) and \(zb_4\) with the key \(kb\) and the open variable \(wb_4\) with the public key of \(A_1\):

\[
\text{join}(B_4, [A_1/\{xb_4, kb/xa_3, kb/zb_4\}], C)
\]

\[
\{ (\text{out}(\{m_1\}_{ka}).\text{in}(\{?ya_1\}_{A^-_1}))], (\text{in}(\{?vb_2\}_{ka}).\text{out}(\{n_2\}_{A^+_3}))],
(xa_3)\text{out}(\{m_3\}_{xa_3}).\text{in}(\{?ya_3\}_{A^-_3})\}, (\text{in}(\{?vb_4\}_{kb}).\text{out}(\{n_4\}_{A^+_3}))\}
\]

### 5.3.3 Operational semantics

The operational semantics of principals executing in a protocol (multi-) session is given, like in the case of interaction patterns, by means of a pair of labeled transition systems: one representing the actions that a principal can execute accordingly to its definition, and a second one representing how the actions done by a principal let him interact with the others.

The “stand alone” behaviour of a principal is modeled by \((\to)\) of Figure 5.2, which is defined up to structural congruence (defined in Figure 3.3) and \(\alpha\)-conversion.\(^2\)

If the principal semantics does not present substantial differences with respect to that one of interaction patterns, the cIP-calculus session semantics differs from interaction patterns session semantics in two relevant points:

\(^2\)The rules are the same of those of pattern semantics of Figure 3.4, but the rules \((\text{silent})\) and \((\text{pat})\) are missing.
1. While in the case of IP-calculus, in order to abstract from possible medium failures, we modeled communications as synchronous actions, in modelling security protocols, where communications happen “in the ether” of a “public” channel that may be accessed by all processes, it is more convenient to model communications as asynchronous actions: a principal outputs a message and another principal, of whose identity is not sure (and hence the first one can not synchronise with it), will get the message from the public channel, possibly after that an intruder has manipulated it. The definition of an asynchronous semantics does not cause any complication. As simplifying, but not less general, assumption, we consider that all the communications are done throughout the intruder (which possibly can forward a received message without modifying it, mimicking a direct communication among two principals).

2. The cryptographic machinery has been introduced without extending the original calculus with new cryptographic actions. This choice, different from most of the proposals in literature, has already been discussed in the introduction of Section 5.3. Cryptography is embedded in communications in the sense that a communication can happen only when sent data match in the expected way the “template” of data that the receiver is waiting for. This means that a (sent) cryptogram can match an input template either if the input action does not attempt any decryption ($in(\{x\})$, for instance), or if it provides the right key for the decryption it is attempting. In the first case the cryptogram is received but the message it contains will not be understood by effect of the input action. In the second case, an input action like $in(\{?x\}^+)$, succeeds in receiving and decrypting the message $\{m\}$. After the successful input/decryption, the message $m$ is assigned to the variable $x$.

Whenever a principal intends to receive and decrypt a datum, it is required to explicitly state its intentions by providing the proper key in an input action, like, for example, $in(\{?x\}^-)$. Notice that this approach does not lose in generality, since principals are expected, in the execution of a protocol, to know the keys they need and when they must use them (and, if not, they could in principle repeatedly send (to themselves) any cryptogram they have acquired and attempt to decrypt it with
one of the keys they possess, by means of a finite number of input actions).

According with the above definitions, expressions of the form \( \text{in}(\{d\}_y) \), which would read as “a principal attempts to receive and decrypt a cryptogram with a key that it still needs to receive”, are not allowed and, as said, binding variable occurrences can not appear as keys.

A suitable definition of matching assures that communication of secrets works in the expected way.

**Definition 5.11 (Matching)** Let \( d \) and \( d' \) be indexed messages in \( M_i \). We say that \( d \) and \( d' \) match \( (d \sim d') \) if and only if

1. if \( d, d' \in PN_i \cup K \cup PN_i^+ \cup PN_i^- \cup NO_i \) then \( d = d' \) \(^3\) or
2. if \( d = (e, f) \) and \( d' = (e', f') \), then \( e \sim e' \) and \( f \sim f' \), or
3. if \( d = \{e\}_\lambda \) and \( d' = \{e'\}_\lambda \) then \( e \sim e' \).

An indexed datum \( d \in M_{Vi} \) matches an indexed message \( d' \in M_i \) if and only if there exists a substitution \( \sigma \) such that \( d\sigma \sim d' \).

Given the notion of mapping, the definition of an asynchronous semantics, where the intruder/environment participate to all the communications, is straightforward, and it is given in Figure 5.3, where communication inside a session and the possible evolution of a session due to the joining of a new instance are modeled.

The transition system is expressed by a relation between states (configurations) \( \langle C, \chi, \kappa \rangle \), where \( C \) is a session, \( \chi \) is a variable binding that keeps track of the associations of the variables due to communications and join executions, and \( \kappa \), the intruder knowledge, contains the names of the instances that joined the session and the data sent along the public channel. Transitions are labeled in order to keep trace of the happened actions.

Rule \((I - in)\) (for Intruder input) states the effects of an output done by a principal. According to our assumptions (every principal is closed and there are not binding occurrence of variables in output actions), a correct principal can output only variable free messages \( (d \in M_i) \). A message \( d \) sent by a principal is recorded in the intruder knowledge \( \kappa \), while the principal evolves as prescribed by principal semantics of Figure 5.2. The label \( i(d) \) records that the intruder has input the message \( d \). The request \( d \in M_i \) prevents ill-formed messages, like, e.g., cryptograms with couples as key, to be communicated.

Rule \((I - out)\) (for Intruder output) states the effects of an input done by a principal. The premise of this rule requires the existence of a message \( m \) which is derivable from \( \kappa \), and it matches, via a ground substitution \( \sigma \), the input datum \( d' \) that a principal in the session is ready to receive. The substitution consists of a set

\(^3\)This equality is syntactical equality, so that indexes are relevant with respect to matching.
of assignments for the variables having a binding occurrence in $d'$. The principal, hence, evolves according to the executed action, as prescribed by principal semantics of Figure 5.2, and $\sigma$ is applied to its continuation. Moreover, $\sigma$ is also recorded in $\chi$, a substitution which faithfully record the communications so far happened in the session. The label $o(m)$ records that the intruder has output the message $m$.

Rule $(join)$ adds a new instance of a principal to the session. It requires a new instance of a principal (of the protocol), $A_i$ say, and a corresponding mapping $\gamma$ which appropriately shares secret keys between $A_i$ and the other principals already in the session. The effects of the application of this rule are that $\gamma$ is recorded in $\chi$ and the name $A_i$, and hence its public key, are added to the intruder knowledge $\kappa$. Moreover, $\gamma$ is applied to all the open variables which occur in the principals. The label $j(A_i, \gamma)$ records that the instance $A_i$ joined the session by means of the mapping $\gamma$.

Notice that a “bad” $\gamma$ may yield a “wrong” session where keys are not shared in the intended way. As said, understanding the correct sharing of keys, or, equivalently, discovering a key distribution which permits an attack to be successfully carried out, is one of the subtleties which make protocol analysis difficult. A strategy to bound the possible choices of keys to be shared will be discussed Section 5.5, where the automatic verification technique will be introduced.

The transition semantics defined characterises the possible computations relative to a given protocol. Starting from an empty session, the set of the states which can be reached, according to the rules of the semantics, constitutes the state space of the possible computations that a multi-session run of the protocol can generate.

An initial state represents a session that does not yet contain any participant. The intruder may be equipped with an initial finite knowledge. A final state represents a session where all the principal instances that joined the session terminate.
Definition 5.12 (Initial and final states) A state \( \langle \{\}, \chi, \kappa \rangle \) is
- initial if and only if \( \chi \) is the empty substitution and \( \kappa \) is a finite subset of \( M \);
- final if and only if \( \chi \) is a substitution and \( \kappa \) is a finite subset of \( M \).

A sequence of states, obtained one from another by the application of the transition rules of the semantics represents one of the possible computation of the session, also called trace (more rules can be applied at the same time, and each rule may be applied with different premises, generating more computations). Only traces starting from initial states are considered. They are terminating if they end in a final state (also indicated as a state with no principals in it).

Definition 5.13 (Trace) A trace is a sequence
\[
T_s = \Sigma_0.\alpha_1.\Sigma_1 \ldots \alpha_n.\Sigma_n
\]
where \( \Sigma_0 \) is an initial state and \( \Sigma_{i-1} \overset{\alpha_i}{\rightarrow} \Sigma_i \) \((1 \leq i \leq n)\). \( T_s \) is terminating if \( \Sigma_n \) is a final state.

Example 5.6 Referring to Example 5.1, let us consider the formalisation of the Denning Sacco Key Distribution protocol, and one of its possible traces. The informal definition of the protocol is

\[
\begin{align*}
(1) \quad & A \rightarrow S : \quad A, B \\
(2) \quad & S \rightarrow A : \quad \{ A, A^+ \}_{S^-}, \{ B, B^+ \}_{S^-} \\
(3) \quad & A \rightarrow B : \quad \{ A, A^+ \}_{S^-}, \{ B, B^+ \}_{S^-}, \{ \{ k \} A^- \}_{B^-}
\end{align*}
\]

and the chosen formalisation, which fixes the intended meaning of the informal language follows (for the sake of simplicity tuple parenthesis are omitted).

\[
\begin{align*}
S \triangleq & \ (\text{in}(\{ x, y \}). \ \text{out}(\{ x, x^+ \}_{S^-}, \{ y, y^+ \}_{S^-})) \\
A \triangleq & \ (xa, ya). \ \text{out}(A, ya). \ \text{in}(\{ A, A^+ \}_{xa^-}, \{ ya, za \}_{xa^+}) \\
B \triangleq & \ (xb). \ \text{in}(\{ wb, yb \}_{xb^+}, \{ B, B^+ \}_{xb^+}, \{ zb \}_{yb} B^-)
\end{align*}
\]

In this formalisation, \( S \) receives by a communication two instance names and provides their public keys “certifying” them (another choice, less adherent to the informal specification, could have required that \( S \) is joined by means of open variables to two instances of which it provides the public keys). \( A \) joins the session by acquiring, by means of open variables, the public key (actually, the name) of a trusted server, \( xa \), and the name of (the instance of) the principal, \( ya \), with which it will share the session key. Notice that in this way it is solved the problem for \( A \) of deciding, in a multi-session run, with which instance of \( B \) it interacts. \( A \), then, waits from \( S \) for the certificates regarding the public key of the selected instance of \( B \) and its own.
public key. Finally, A sends the session key $k$ appropriately encrypted. B needs the public key of $S$ (open variable $xb$), and then it can receive the name, $yb$, and the corresponding public key, $yb^+$ of an instance of A, and, finally, the cryptogram containing a session key $k$.

The protocol has the goal of certifying the public keys of $A$ and $B$ via a trusted server whose public key is known. According to this interpretation, in the proposed formulation a public key, $A^+$ say, can not be derived by the knowledge of the corresponding principal name $A^+$, except for the case of the trusted server. The coincidence of names and public keys, broadly adopted, appears reasonable as soon as one may wish to abstract from the mechanism of distribution of public keys and may wish to assume that public keys are available to all the principals. Moreover, it is also reasonable to assume that an intruder knows all the public keys of the instances in the session, (or, equivalently, that an attack can not be prevented by hiding a public key to the intruder).

It must be noticed that this formalisation does not fully respect the assumptions we made about the allowed syntax of the language since $A$ “uses” the private key of $S$, namely $xa^-$, in its last action $\text{out}((\{A, A^+\}_{xa^-}, \{B, za\}_{xa^-}, \{\{k\}_{A^-}\}_{za})$. This is due to the fact that $A$ needs to decrypt the message in order to get the public key of $B$, and in doing that the message is decomposed in its parts. At this point, namely, $A$ does not possess the original message and could not forward it. This “controlled” use of the private key of another principal is not harmful since it does not increase the capabilities of a principal.\(^4\)

The trace relative to a session joined in the intended way by one instance of each principal is as follows. The initial state is:

$$\Sigma_0 = (\{\}, \varepsilon, \emptyset)$$

Let us assume, among the possible choices, the first three transitions from the initial state are due to the joining of the principal instances:

$$\Sigma_0 \xrightarrow{\text{in}(\{?x_s1, ?y_s1\). \text{out}((\{xs_1, xs_1^+\}_{S_1^-}, \{ys_1, ys_1^+\}_{S_1^-}))}, \Sigma_3,$$

where

$$\Sigma_3 = \langle \text{in}(\{\text{out}(A_2, B_3). in((\{A_2, A_2^+\}_{S_3^+}, \{B_3, ?za_2\}_{S_3^+}), \{\{k\}_{A_2^-}\}_{za_2}), \text{out}((\{A_2, A_2^+\}_{S_3^-}, \{B_3, za_2\}_{S_3^-}, \{\{k\}_{A_2^-}\}_{za_2}), \text{in}((\{?za_3, ?yb_3\}_{S_3^+}, \{B_3, B_3^+\}_{S_3^+}, \{\{za_3\}_{ya_3}\}_{B_3^-}), \{xa_2 \rightarrow S_1, ya_2 \rightarrow B_3, xb_3 \rightarrow S_1\}), \{S_1, A_2, B_3\} \rangle$$

\(^4\)Actually, it could be avoided using a more complex behaviour, where the principal firstly receives the message as is, linking it to a variable. The principal can then forward the original message and receiving it back, decrypting it, like in $\ldots \text{in}(\{?ja\). \text{out}(ja). \text{in}((\{A, A^+\}_{xa^-}, \{B, za\}_{xa^-}, \{\{k\}_{A^-}\}_{za})), \ldots$ Note that the extra input/output actions introduced can not be exploited by the intruder for enhancing its knowledge. The chosen formalisation simplifies the understanding of the example.
The first communication action could be either an input (by the intruder) of the only message that can be sent at the present by $A_2$, or an output of the intruder with any (of the infinitely many) couple that can be derived from the current $\kappa$ and sent to $S_1$, or to $B_3$ if properly encrypted with the available public key of $S_1$. Let us follows the intended trace of the protocol with the above boldface action.

$$\Sigma_3 \overset{i(A_2,B_3)}{\mapsto} \Sigma_4 = \langle \{()\text{in}(\langle xs_1, ys_1 \rangle), \text{out}(\{xs_1, xs_1+\}_{S_1^-}, \{ys_1, ys_1+\}_{S_1^-})\},$$

$$()\text{in}((\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-})).$$

$$\text{out}(\{A_2, A_2^+\}_{S_1^-}, \{B_3, za_2\}_{S_1^-}, \{\{k\}_A_{2^-}^{+} za_2\}),$$

$$()\text{in}(\{?wb_3, ?yb_3\}_{S_1^{+}}, \{B_3, B_3^+\}_{S_1^{+}}, \{?zb_3\}_{yb_3 B_3^+}),$$

$$[xa_2 \rightarrow S_1, ya_2 \rightarrow B_3, xb_3 \rightarrow S_1],$$

$$\{S_1, A_2, B_3, (A_2, B_3)\} \rangle$$

From $\Sigma_4$, according to the informal specification of the protocol, the input action of $S_1$ chosen and executed. Note that, in this choice is also the only one possible, in fact, due to the secrecy of $S_1^-$, no cryptogram matching the input actions of $A_2$ or $B_3$, i.e. $\{A_2, A_2^+\}_{S_1^-}$ and $\{A_2, ..., A_2\}_{S_1^-}$, can be generated from the current $\kappa$.

$$\Sigma_4 \overset{o(A_2,B_3)}{\mapsto} \Sigma_5 = \langle \{()\text{out}(\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-})\},$$

$$()\text{in}((\{A_2, A_2^+\}_{S_1^-}, \{B_3, za_2\}_{S_1^-})).$$

$$\text{out}(\{A_2, A_2^+\}_{S_1^-}, \{B_3, za_2\}_{S_1^-}, \{\{k\}_A_{2^-}^{+} za_2\}),$$

$$()\text{in}(\{?wb_3, ?yb_3\}_{S_1^{+}}, \{B_3, B_3^+\}_{S_1^{+}}, \{?zb_3\}_{yb_3 B_3^+}),$$

$$[xa_2 \rightarrow S_1, xb_3 \rightarrow S_1, xs_1 \rightarrow A_2, ys_1 \rightarrow B_3],$$

$$\{S_1, A_2, B_3, (A_2, B_3)\} \rangle$$

From $\Sigma_5$, $\Sigma_8$ is reached by the three transitions below (corresponding to the above boldface actions):

$$\Sigma_5 \overset{i(A_2,A_2^+)}{\mapsto} \Sigma_8 = \langle \{()\text{in}(\langle wb_3, yb_3 \rangle), \text{out}(\{wb_3, yb_3\}_{S_1^{+}}, \{?wb_3\}_{yb_3 B_3^+}),$$

$$[xa_2 \rightarrow S_1, xb_3 \rightarrow S_1, xs_1 \rightarrow A_2, ys_1 \rightarrow B_3, za_2 \rightarrow B_3^+],$$

$$\{S_1, A_2, B_3, (A_2, B_3), (\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-}),$$

$$\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-}, \{\{k\}_A_{2^-}^{+} B_3^+\} \rangle \rangle  \mapsto \Sigma_8$$

where

$$\Sigma_8 = \langle \{()\text{in}(\langle wb_3, yb_3 \rangle), \text{out}(\{wb_3, yb_3\}_{S_1^{+}}, \{?wb_3\}_{yb_3 B_3^+}),$$

$$[xa_2 \rightarrow S_1, xb_3 \rightarrow S_1, xs_1 \rightarrow A_2, ys_1 \rightarrow B_3, za_2 \rightarrow B_3^+],$$

$$\{S_1, A_2, B_3, (A_2, B_3), (\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-}),$$

$$\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-}, \{\{k\}_A_{2^-}^{+} B_3^+\} \rangle \rangle  \mapsto \Sigma_8$$

From $\Sigma_8$, in one step a final state is reached:

$$\Sigma_8 = \langle \{()\text{in}(\langle wb_3, yb_3 \rangle), \text{out}(\{wb_3, yb_3\}_{S_1^{+}}, \{?wb_3\}_{yb_3 B_3^+}),$$

$$[xa_2 \rightarrow S_1, xb_3 \rightarrow S_1, xs_1 \rightarrow A_2, ys_1 \rightarrow B_3, za_2 \rightarrow B_3^+],$$

$$\{S_1, A_2, B_3, (A_2, B_3), (\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-}),$$

$$\{A_2, A_2^+\}_{S_1^-}, \{B_3, B_3^+\}_{S_1^-}, \{\{k\}_A_{2^-}^{+} B_3^+\} \rangle \rangle  \mapsto \Sigma_8$$
Concluding the presentation of the semantics, it is important to remark that, as is, it is not effective, in the sense that it has not been specified how the message $m$ of rule $(I-out)$ can be chosen. One might expect that the choice of a message $m$ to be derived from $\kappa$ can be driven by the input that a principal is ready to receive.

For instance in the previous example, the expected input data of principals $S_1$, $A_2$ and $B_3$ have driven the selection of the messages derived from $\kappa$ (done also according to the expected functioning of the protocol) and used in output transitions. More specifically, in the presented trace the intruder simply forwards the messages it receives without modifying them, and hence deriving a message from $\kappa$ reduces to testing its membership.

In general, if a principal is waiting for a not specified message, i.e. a variable, or for a message containing a not specified submessage, like $\gamma y$ in $in(\{\gamma y\}_k)$, every message that can be derived from $\kappa$ could be sent, and none of them can be ignored, since it could be the (only) one which permits the attack to be performed.

In presence of data constructors, the messages that can be derived from a not empty set $\kappa$ are infinite. For example, from a set containing only a symmetric key $\kappa = \{k\}$, within a language provided with encryption and pairing, the messages $\{k, (k, k), \{k\}_k, (k, k, k), \{\{k\}_k, \ldots\}$ can be derived.

The presented semantics does not deal with the infinite branching problem that may happen when, in applying rule $(I-out)$, an infinite number of transitions (one for each different message) can be generated. In Section 5.7 we will show how symbolic analysis techniques deal with an infinite number of possible messages, and allow us to define a finite branching symbolic semantics which is “coherent” with respect to this concrete, but infinitely branching, semantics.

### 5.4 \(PL\) Logic: expressing security properties

This section presents \(PL\) (protocol logic), a logic for specifying properties of security protocols. The logic is defined according to the idea that the properties of interest can be formulated in terms of the values that are exchanged in the protocol, the relations among the principals which exchange these data, and the amount of the knowledge that an intruder can acquire during the execution of a protocol. Moreover, since we intend to verify protocols against multi-session attacks, the logic must permit different principal instances, and the respective data, to be distinguished.

For these reasons, the logic allows one to state the values that variables must assume and the values that must or must not belong to $\kappa$. Relations between principal instances are expressed by introducing index variables in order to predicate also over the instance indexes. Index variables range over instance labels. In this way, data are associated to the instances in which they occur, and (causal) relations over data, expressed by the formulas, induce relations over the instances in which data occur.

Since formulas concern dynamic aspects, like relations among variables and the
values they assume, the satisfaction of a formula depends on the point in the execution of a (multi-)session of the protocol in which it is tested. Information about a point in an execution trace is resumed by $\chi$ and $\kappa$ of the states of protocol semantics (see Section 5.3.3), i.e. by the assignments done to the variables of principal instances, and the knowledge acquired by the intruder, respectively. Models of formulas are thus defined in terms of $\chi$ and $\kappa$. Once that a property has been formalised by a formula, the protocol enforces the property if the formula holds for all the possible traces of the protocol.

In this setting, integrity is expressed by a property like “a datum can not differ from its expected value”, generalizing the approach introduced in [AG99]. Secrecy is handled by exploiting intruder knowledge and the values it may or may not contain. As far as authentication is concerned, many authors reduce it to causality relations over the structure of the events observed in the state space of the computation. A common approach is to exploit correspondence relation [CJM98, Bor01], that can express several forms of causality, like, e.g., “the first message sent by a responder has always been preceded by a 'corresponding' message sent by an initiator of the protocol —and hence the responder is sure of the identity of the initiator”. This kind of causal relationships can be expressed, in our logic, in terms of implications regarding the values assumed by indexed variables, and, consequently, by the instances in which such variables occur.

### 5.4.1 Syntax

A distinguishing feature of the logic is the use of index variables ranging over the labels (numbers) of the instances in a session. If $xa$ is a variable occurring in principal $A$ (reminding that the name of the variable indicates the principal in which it occurs), then $xa_i$ ranges over the instances of $xa$ occurring in the instances of the principal $A$, and $i$ is an index variable. Indexed variables are quantified by the quantifiers $\forall i : A$ and $\exists i : A$, which are read respectively as “for all the instances $i$ of $A$” and “exists an instance $i$ of $A$”.

Intuitively speaking, a formula like $\exists i : A. \kappa \ni xa_i$ states that exists an instance of $A$, say $A_3$, containing the variable $xa_3$, and that the value associated (in a trace) to the variable belongs (at the end of the trace) to the intruder knowledge $\kappa$. $^5$ It is possible to denote both different variables occurring in a principal, by means of their names, and different instances of the same variable, by means of their index. For instance, the formula $\forall i : A. xa_i \neq ya_i$ states that for each instance of $A$, the two variables $xa$ and $ya$ never assume the same value, while $\forall i : A. \forall j : A. xa_i \neq ya_j$ states that it never happens that any two instances of $xa$ and $ya$ assume the same value, whichever instance of $A$ they may occur in.

$^5$Note the difference between the meta-variable $i$, as it has been used up to now to indicate a generic instance of $A$, like in $A_i$, and the index variable $i$, belonging to the logic language, in the formula $\exists i : A. A_i \in \kappa$. Since the context always distinguishes between the two cases, we used, for the sake of simplicity, the same notation for both of them.
Moreover, index variables are also associated to data, so as to express statements like \( \forall i : A. A_i \not\in \kappa \): “the intruder does not know any of the private keys of the instances of \( A \)”. The data which do not belongs to a given instance (global names), instead, must not be indexed. This is the case of symmetric keys shared between more principals and possibly even between instances of the same principal, like, e.g., in \( \forall i : A. xa_i = k \) which reads as “all the instances of \( A \) associate to the variable \( xa \) the same symmetric key \( k \)”. Indicating with \( V_I \) the set of variables labeled with index variables, and with \( M_{VI} \) the set of data labeled with indexed variables, the syntax of the logic is given by the following definition.

**Definition 5.14 (\( \mathcal{PL} \) – Syntax)** Let \( m \in M_{VI}, \ xa_j \in V_I \) and \( A \in PN \). The formulas \( \phi, \psi \) of the logic \( \mathcal{PL} \) (protocol logic) are defined as follows:

\[
\phi, \psi ::= xa_i = m \mid \kappa \triangleright_{PL} m \mid \forall i : A. \phi \mid \exists i : A. \phi \mid \neg \phi \mid \phi \wedge \psi \mid \phi \vee \psi
\]

Logic consists of equalities among a variable and a datum (which in turn may contain variables), the derivability operator \( \triangleright_{PL} \): \( \kappa \triangleright_{PL} m \) reads as “message \( m \) can be derived by the knowledge \( \kappa \)”,\(^6\) universally and existentially quantified formulas. Semantics will make clear how the models of a quantified formula, like, e.g., \( \forall i : A. \phi \), associate to the variable \( i \) “correct” labels (natural numbers), i.e. labels corresponding to the indexes of the instances of \( A \) which have actually participated to the protocol session.

The standard boolean operators complete the definition of the logic, allowing for more complex statements, like “if two principals share a given key, then they can not acquire, (as the protocol session evolves), a key shared by others principals”, or “if a principal \( A \) receives a message from the principal \( B \), then a reference to the sender occurring in the knowledge of \( A \) (for example, its public key) must refer exactly to \( B \)”.

The derived formulas \( \phi \rightarrow \psi, \ xa_i \neq m \) and \( m \not\in \kappa \) read as usual as \( \neg \phi \vee \psi, \neg(xa_i = m) \) and \( \neg(m \triangleright \kappa) \), respectively.

Note that the presented syntax allows for “inconsistent” formulas, like, e.g. \( \exists i : A. xc_i = B_j \), when the variable \( xc \) do not occur in \( A \). Rather than further specifying the set of “legal” formulas, we consider this a wrong specification of a security property, which may lead to an erroneous verification.

**Example 5.7** This more structured example concludes the presentation of the syntax of \( \mathcal{PL} \), by illustrating two simple properties which regard the protocol specified in Examples 5.4 and 5.5. The first one says that the symmetric key used by any instance of \( A \) should be kept secret, i.e. it should not be known by the intruder (remember that this formula will be verified with respect to a termination state, where \( xa_i \) has been assigned to a message):

\[
\forall i : A. xa_i \not\in \kappa.
\]

\(^6\)The symbol \( \triangleright_{PL} \) has been labeled with \( PL \) in order to distinguish the operator of the logic from the relation symbol for the derivation of a message from a knowledge. Whenever the context clearly distinguishes them, \( \triangleright \) will be used for both of them.
The second one expresses a relation among data sent by instances of A and B in the protocol. Whenever B_i receives a message m_j from A_j, it then sends back the nonce n_i to the same instance A_j. Formally:

\[ \forall i : B. \exists j : A. vb_i = m_j \rightarrow ya_j = n_i. \]

### 5.4.2 Models for PL

Satisfaction of formulas depends on the way in which keys are shared and data are exchanged accordingly to the interaction that has taken place during a trace of a session of the protocol. It, hence, depends on the assignments done, the instances that actually have participated in the computation, and the knowledge acquired by the intruder throughout a trace until the point in which formulas are checked. As said, the information of a state \( \langle C, \chi, \kappa \rangle \) which is relevant for the satisfaction of a formula consists of the assignment \( \chi \) and the knowledge \( \kappa \).

More precisely, abstracting from the traces which they belong to, the class of the models for the PL logic consists of pairs \( \kappa, \chi \). The notation \( \kappa \models \chi \phi \), then, reads as “the set \( \kappa \), under the variable assignment \( \chi \), is a model for the formula \( \phi \).”

Satisfaction, introduced in the following definition is given for closed formulas \( \phi \), which do not contain free index variables.

**Definition 5.15 (Model for PL formulas)** Let \( \chi \) be a substitution from variables \( V_i \) to indexed messages \( M_i \) and \( \phi \) a formula of PL. Then a knowledge \( \kappa \) is a model for the formula \( \phi \) under to the substitution \( \chi \) if \( \kappa \models \chi \phi \) can be proved by the following rules (where \( n \in \mathbb{N} \) is an instance label):

\[
\begin{align*}
\frac{xa_n \chi = m \chi}{\kappa \models \chi xa_n = m} \quad \text{(=)} & \quad \frac{\kappa \triangleright m \chi}{\kappa \models \chi \kappa \triangleright m} \quad \text{(\( \triangleright \))} \\
\frac{\exists n. A_n \in \kappa}{\kappa \models \chi \exists i : A. \phi} \quad \text{(\( \exists \))} & \quad \frac{\forall n. A_n \in \kappa}{\kappa \models \chi \forall i : A. \phi} \quad \text{(\( \forall \))}
\end{align*}
\]

\[
\begin{align*}
\frac{\kappa \models \chi \phi \quad \kappa \models \chi \psi}{\kappa \models \chi \phi \wedge \psi} \quad \text{(\( \wedge \))} & \quad \frac{\kappa \models \chi \phi}{\kappa \models \chi \phi \lor \psi} \quad \text{(\( \lor \))} \quad \frac{\kappa \models \chi \psi}{\kappa \models \chi \phi \lor \psi} \quad \text{(\( \lor \))} & \quad \frac{\kappa \models \chi \phi}{\kappa \not\models \chi \neg \phi} \quad \text{(-)}
\end{align*}
\]

Note that the relation \( \kappa \models \chi \psi \) is defined for formulas \( \psi \) which are a superset of the formulas of LP, in fact \( \models \) is defined also for formulas whose index variables have been instantiated with instance labels \( n \). This is necessary in order to verify a formula against the actual instances which have participated to the protocol.

An equality \( xa_n = m \) holds if \( \chi \) maps the variable of the principal instance into the datum \( m \chi \), where the variables which possibly occur in \( m \) have also been instantiated by \( \chi \).
The formula $\kappa \triangleright m$ holds, if the datum $m\chi$, where the variables of $m$ have been instantiated by $\chi$, can be derived from $\kappa$ (see Section 5.2 for the definition of message derivation).

Quantifiers are resolved by substituting index variables with instance labels. Labels can be chosen among the ones of the instances of the principal declared by the quantifier ($i : A$) and must belong to instances which have been involved in the protocol session ($A_n \in \kappa$). In the case for the existential quantifier, the formula holds if one of the formulas instantiated with one of possible labels holds. In the case for the universal quantifier, the formula holds if all the formulas instantiated with each one of possible labels hold. In both cases the number of possible labels is finite.

Since formulas are requested to be closed, all the index variables of a formula fall in the scope of a quantifier and are hence necessarily substituted with instance labels by the rules for quantifiers. This justifies rules $=\!\!=$ and $\triangleright\!\!\triangleright$ defined for formulas in which index variables have been replaced by labels.

Rules for $\lor$ and $\land$ operators are standard. About the rule for $\neg$, note that the inductive definition given above is based on a close world assumption, in the sense that $\kappa \models \chi \phi$ 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 “$\kappa$ is a model for $\neg \phi$ with respect to $\chi$ if $\kappa$ is not a model for $\phi$ with respect to $\chi$”, i.e. if does not exist a proof for $\kappa \models \chi \phi$. Since relations $\triangleright$ and $\models$ are decidable (and the complexity of formulas decreases at each rule application), it follows that $\models$ and, hence, $\not\models$ are decidable too.

If two substitutions $\chi$ and $\chi'$ differ on variables not appearing in $\phi$, then $\kappa \models \chi \phi \Leftrightarrow \kappa \models \chi' \phi$; hence, we can only consider finite assignments over the variables of $\phi$.

The following two examples show a case of wrong-defined property and a proof for the validity of a formula with respect to a trace.

**Example 5.8** Given the session:

\[
\{ (\text{out}(\{k\}_k).\text{in}(x_{a_1}), (\text{in}(?y_{b_2}).\text{out}(y_{b_2})) \},
\]

containing principals $A_1$ and $B_2$, the formula $\exists i : A. x_{a_i} = ka \rightarrow y_{b_i} = k$ does not hold in any of its possible traces. In fact, there is not any instance of a principal $A$ containing a variable $y_{b_i}$ as requested by the scoping of the existential quantifier. The formula has not, informally speaking, a “coherent” meaning with respect to the instances of the session. Coherent formulas must have a coherent mapping of variable indexes to actual instance labels. Formally, the only label for $i$ is 1 and the equality $y_{b_1} = k$ can not be made true by any $\chi$ resulting from the session.

**Example 5.9** Let us consider the second property of Example 5.7 expressed as $\phi \equiv \forall i : B. \exists j : A. \neg(v_{b_i} = m_j) \lor y_{a_j} = n_i$. Let $\chi$ be the assignments $\{v_{b_1} \mapsto m_2, y_{a_2} \mapsto \ldots\}$.
5.5. CONCRETE PROTOCOL VERIFICATION

Having defined how to formally specify a security protocol, the property it is supposed to guarantee, and having precisely described the capabilities of a malicious intruder, it is now possible to define the notion of attack to the protocol and show how to verify the existence of an attack within the execution of a multi-session run of the protocol.

An attack simply consists of a trace of the protocol in which the desired property does not hold. In general, satisfaction of formulas can be checked in significant states of the computation, while in our framework verification is carried on by checking if an intruder is able to drive the session to a successful terminating state where the desired property does not hold (and hence the protocol is unsafe). Intuitively speaking this is motivated by two main reasons:

1. we consider an attack which prevents any of the instances in the session from successfully terminate as an execution error, like, for example, the failures due to communication medium errors. Instead, we are interested in discovering those attacks in which the intruder can spoil the protocol without that the principals are aware of it (and hence they must at least terminate),

2. we believe that most significant protocol security failures are monotonic properties of a trace, in the sense that if a security property is violated at a certain state of a trace, it will not be satisfied at the successful termination of the same trace.

**Definition 5.16 (Attack)** Given a security protocol, an initial knowledge $\kappa_0$ provided to the intruder, and a security property $\phi$, then a terminating trace of the protocol

$$T_s = \langle \{\}, \epsilon, \kappa_0, \alpha_1, \ldots, \alpha_n, \langle \{\}, \chi, \kappa \rangle \rangle$$

is an attack if and only if $\kappa \not\models_\chi \phi$ (or, equivalently, $\kappa \models_\chi \neg \phi$).

The procedure for protocol verification consists of three steps:
• **Protocol formalisation.** As discussed, protocols are generally presented in an informal language. Translating such an informal specification into a formal CSP specification requires some choice to be done and is a non-automated step. A good translation, faithfully respecting the “intended” functioning of the protocol is the first step required by protocol verification.

To have an idea of how much the functioning of the protocol can be influenced by apparently insignificant choices about the “unsaid” assumptions of the informal specification, consider the very simple protocol

\[
(1) \quad A \rightarrow B : \{k\}_k.
\]

The straightforward translation

\[
A \triangleq [\text{out}(\{k\}_k)] \\
B \triangleq [\text{in}(\{?xb\}_k)]
\]

states that all the possible instances of \(A\) and \(B\) share the same key \(k\), while the translation

\[
A \triangleq (ya)[\text{out}(\{k\}_{xa})] \\
B \triangleq (yb)[\text{in}(\{?xb\}_{yb})]
\]

requires, more coherently with the kind of protocol, that different instances must be provided with a (session) symmetric key, before they can interact.

• **Property formalisation.** The difficulties and the needed expertise for exactly characterizing the property that the protocol is expected to guarantee have already been discussed. This is also a non-automated step.

• **Search for an attack.** The tedious and error prone phase of attack search can be automated under the following hypothesis:

  - **Infinite branching.** It has been discussed how the the infinite number of messages that can be generated by an intruder and sent to one of the principals in the session causes infinitely branching traces, making state exploration not effective (it is not decidable whether an attack exists). This problem will be addressed in Section 5.7.

  - **Maximum number of sessions analysed.** It is known in the literature that security properties do not necessarily scale up with the number of interleaved sessions, and hence a protocol proved secure for \(n\) sessions may be attacked in exploiting a bigger number of sessions, see, e.g., [Mil99]. It resulted very difficult to find conditions under which it is safe to bound the analysis to a finite number of sessions, and the obtained results do not hold for the general case, see, e.g., [Low98].
Also in our case, the search for an attack diverges by incrementally checking the executions of the protocol with an increasing number of interleaved sessions. This problem is typically addressed by explicitly considering a maximum number of principal instances [MMS97, Hui99, Bor01]. It is worth noticing that thanks to the quantification over instances embedded in the logic, properties can be specified independently of the number of sessions one may wish to checked, so that the maximum number of instances can be flexibly treated as an implicit parameter.

- **Initial set of (symmetric) keys.** Every time a (join) rule is applied in a trace, a $\gamma$ assignment, mapping open variables into keys, must be provided. Open variables for asymmetric keys (Definition 5.10) are mapped in the private or public keys of the instances which are already in the session. Since their number is finite, the number of possible resulting transitions is finite, too.

On the other hand, instances could share any of the infinitely many public keys. According with the vast majority of the proposals in the literature, we assume that a finite set of symmetric keys may circulate in a session, and this set is provided within the initial knowledge of the intruder. The number of different keys can be reasonably bounded by the maximum number of open variables occurring in the instances of the current number of sessions under analysis.

The attack search falls in the class of model checking techniques, [CW+96], and consists in a trace state exploration. At each step, an increasing number of session is checked. For a given number of sessions all the traces are developed. Note that all the possible choices (except the generation of messages in the ($I-out$)rule), i.e. the action to perform, the principal to join and the mapping used in the join, are done over a finite number of possible choices.

The process is potentially unbound in the number of session checked. As said, this is an unavoidable fact, and termination can be forced by fixing a maximum number of sessions to be checked.

The exhaustive state exploration returns a (finite) set of tentative models for the formula representing the property. As shown, verification of the formula is also a decidable (finite) process. All the models for the formula are attacks for the protocol.

Informally speaking, the absence of attacks, together with the exhaustive state exploration, guarantees that

- under the interpretation of the algorithm consisting of its formalisation in cIP-calculus,

- under the interpretation of the desired formula consisting of its formalisation in $\mathcal{PL}$ logic,
– up to the number of sessions checked,  
the protocol is correct.

The sketched procedure allows for several optimisations when implemented, in order to drive the possible choices towards a faster convergence to the “interesting states”. We cite one of the most promising among those under study, i.e. the possibility of giving priority to the (symbolic) messages that seems heuristically more promising with respect to the satisfaction of the formula (a sort of back propagation of the constraints that the formula imposes in order to be (not to be) satisfied).

5.6 An Oracle discovers an attack

The proposed framework is shown at work, while detecting an attack to the Denning Sacco Key Distribution [CJ97] protocol. The protocol has been presented in Example 5.1, and formalised in Example 5.6, where a trace regarding a single-session execution has been illustrated. The protocol consists of the following steps:

(1) \( A \to S : \ A, B \)
(2) \( S \to A : \ \{A, A^+\}_{S^-}, \{B, B^+\}_{S^-} \)
(3) \( A \to B : \ \{A, A^+\}_{S^-}, \{B, B^+\}_{S^-}, \{\{k\}_{A^-}\}_{B^+} \)

and it is intended to let \( A \) and \( B \) share a symmetric key, being \( B \) sure of \( A \)’s identity, as certified by the trusted server \( S \).

The first step of the analysis, formalisation of the protocol, has already been reported in Example 5.6 and is only briefly recapped. In the second one, the property the protocol is supposed to enforce is formalised as a \( \mathcal{PL} \) formula. The third one, attack search, is carried out under the supervision of an oracle, which “guesses,” among the infinitely many, the correct messages to be sent by the intruder in order to succeed in the attack.

Protocol formalisation

The formalisation of the protocol has been given Example 5.6, where it is more carefully explained. It is anyway worth recalling that each principal instance gets the name and the public key of a partner instances by means of the open variables and that the session key \( k \) is private (local name) of each instance of \( A \) (they must be distinguished as if they were different names, but only one of them occurs in the attack of the example, and hence the only name \( k \) is used).

\[
S \triangleq (\{in(\?xs, \?ys). out(\{xs, xs^+\}_{S^-}, \{ys, ys^+\}_{S^-})\})
\]

\[
A \triangleq (xa, ya)[out(A, ya). in(\{A, A^+\}_{xa^-}, \{ya, ?za\}_{xa^+})].
\]

\[
out(\{A, A^+\}_{xa^-}, \{ya, za\}_{xa^-}, \{\{k\}_{A^-}\}_{za})
\]

\[
B \triangleq (xb)[in(\{?wb, ?yb\}_{xb^-}, \{B, B^+\}_{xb^+}, \{\{?zb\}_{yb}\}_{B^-})]
\]
Property formalisation

The protocol has been designed to guarantee both the secrecy of the session key $k$, and the identity of instances of $A$. Hence, both secrecy and authentication properties are involved. More precisely, after a successful execution of the protocol, $B$ is supposed to be sure with whom it is talking to and that they both secretly share the key $k$.

The secrecy of $k$ is straightforwardly represented by the formula $\forall i : B. \kappa \not\in z_{i_B}$, while authentication is, as usual, more difficult to be properly formalised. One might be tempted to relate $A$ and $k$, according to the viewpoint of $B$, by a property like $\forall i : B. w_{i_B}^+ = y_{i_B}$. The property is read as “$k$ has been encrypted by a principal instance, which, as certified by a server, intended to start a communication”. Unfortunately, there is no guarantee that the above mentioned instance actually intended to communicate with the instance of $B$. In fact, as seen in the attack shown in Example 5.1, the tuple $\{A, A^+\}_{S^-}, \{B, B^+\}_{S^-}, \{\{k\}_{A-}\}_{B^+}$, not being protected by any encryption, can easily be manipulated so as to convince $B$ that $A$ intends to communicate with it.

A correct formulation also models the intentions of $A$, so as to match together the instance of $A$, which starts the protocol, and the instance of $B$ with whom $A$ intended to communicate. In other words, if, from the data received by $B$ it is possible to deduce that an instance of $A$ started a communication, it must also be verified that the instance of $A$ intended to communicate exactly with that instance of $B$:

$$\forall i : B. \exists j : A. y_{i_B} = A_j \rightarrow y_{A_j} = B_i.$$  

From the viewpoint of any $B_i$, $y_{i_B}$ is the private key of an instance $A_j$ encrypting $k$. Then the instance $A_j$ must be the one which acquired the public key of $B_i$ and asked the server to certify the corresponding communication.

The overall secrecy and authentication property relative to the Denning Sacco Key Distribution protocol is hence represented by the following formula:

$$\phi = \forall i : B. \kappa \not\in z_{i_B} \land \forall i : B. \exists j : A. y_{i_B} = A_j \rightarrow y_{A_j} = B_i.$$  

Attack discovery

The aim of this section is to show how the attack illustrated in Example 5.1 can be automatically detected. The initial knowledge $\kappa_0$ contains the public and private keys of the intruder, $I^+$ and $I^-$, so that the other instances joining the session can be connected to it. We remind that, since this is a key distribution algorithm based on the identity certification done by trusted servers, we can not assume that a public key of an instance can be deduced by its name (except for the trusted servers). Symmetric keys are not needed, since instances do not connect with each other by means of symmetric keys and each $A$ instance has its own (local) symmetric key.
The attack consists of two sessions, where the intruder plays the part of $B$ in one of them and of $A$ in the other one. Two server instances are used, one for each session.

The initial state is

$$
\Sigma_0 = \langle \{\}, \varepsilon, \{I_0, I_0^+\} >
$$

Initially, an instance $S_1$ of the server joins the session, then $A_2$ is connected to the server and to the intruder, by means of its open variables $xa_2$ and $ya_2$, respectively. The instance $B_3$ and a second server $S_4$, to which $B_3$ is connected, finally join the session.

Differently from Example 5.6, $A_2$ has been connected to the intruder, letting it play the part of the responder $B$. This is possible since the public key $I^+$ of the intruder is one of the legal keys known in the protocol that instances can use to be connected to each other (generally, all the attack of the kind “a man in the middle” are based on exploiting, in another session, the information got by playing a role in a session — also note in the following, how the server must certify the public key of the intruder, as a “trusted” principal, in order to let the attack succeed).

Finally, note how it is possible that a principal joins the session and only later its open variable are instantiated (when its partner has also joined the session), like in the case of $B_3$.

$$
\Sigma_0 \xrightarrow{j(S_1:z)} j(A_2; [xa_2 \rightarrow S_1, ya_2 \rightarrow I_0]) j(B_3; \varepsilon) j(S_4; [xb_3 \rightarrow S_4]) \Sigma_4 = \Sigma_9
$$

Communications between $A_2$ and $S_1$, boldface above, work as in Example 5.6, but for the difference of the new partner of $A_2$:

$$
\Sigma_4 \xrightarrow{i(A_2, I_0, I_0) \rightarrow (A_2, A_2^+) \rightarrow (I_0, I_0, I_0) \rightarrow (I_0, I_0, I_0) \rightarrow (I_0, I_0, I_0) \rightarrow (I_0, I_0, I_0) \rightarrow \Sigma_9}
$$

Since $B_3$ is waiting for a message in which it figures as the responder, as witnessed by its name and key in the message, the intruder needs the public key of $B_3$ in order to be able to forge a valid message for $B_3$, and hence, at the present, $I_0$ can not
communicate with $B_3$. The intruder then, guided by the oracle, asks $S_4$ to certify a communication among itself, as $A_2$, and $B_3$: $\Sigma_9 \overset{o(A_2,B_3)i(A_2,A_2^+,B_3,B_3^+,S_4^+)}{\longrightarrow} \Sigma_{11}$:

$$\Sigma_{11} = \langle \{0\}, \emptyset \rangle, \langle \{?wb_3, ?yb_3\}_{S_4^+}, \{B_3, B_3^+\}_{S_4^+}, \{\{?zb_3\}_{y_b_3}\}_{B_3^+} \rangle, \langle \{0\}, \emptyset \rangle, \{xa_2 \rightarrow S_1, y_0 \rightarrow I_0, xb_3 \rightarrow S_1, xs_1 \rightarrow A_2, ys_1 \rightarrow I_0, za_2 \rightarrow I_0^+, xs_4 \rightarrow I_0, ys_4 \rightarrow B_3 \}, \{I_0, I_0^+, I_0^-, S_1^+, A_2, B_3, S_4, S_4^+, (A_2, I_0), (\{A_2, A_2^+\}_{S_4^+}, \{I_0, I_0^+\}_{S_4^+})\}, \langle \{A_2, A_2^+\}_{S_4^+}, \{I_0, I_0^+\}_{S_4^+}, \{\{k\}_{A_2^-}I_0^+\}, (A_2, A_2^+ s_4^+, B_3, B_3^- s_4^-) \rangle \rangle$$

The oracle chooses then the correct message to be sent to $B_3$ in order to let believe the intruder is $A_2$, and they exclusively share the key $k$, namely the message $(\{A_2, A_2^+\}_{S_4^+}, \{B_3, B_3^+\}_{S_4^+}, \{\{k\}_{A_2^-}B_3^+\})$. After that action, a final state is reached in which $\phi$ does not hold and the protocol has been successfully attacked. This section ends by showing both the proof for the derivability of the message, and the proof that the current $\kappa$ and $\chi$ are a model for $\phi$.

The explicit form (Definition 5.4) of $\kappa_{11}, e(\kappa_{11})$, is the following:

$$\kappa_{11} \cup \{\{A_2, A_2^+\}_{S_4^-}, \{I_0, I_0^+\}_{S_4^-}, \{\{k\}_{A_2^-}I_0^+, \{A_2, A_2^+\}_{S_4^-}, \{B_3, B_3^+\}_{S_4^-}, A_2, \{k\}_{A_2^-}, k, B_3^+\}$$

and the proof for $e(\kappa_{11}) \ni_i (\{A_2, A_2^+\}_{S_4^-}, \{B_3, B_3^+\}_{S_4^-}, \{\{k\}_{A_2^-}B_3^+\})$ is (where $\kappa$ stands for $e(\kappa)$):

$$\kappa \ni_i \{A_2, A_2^+\}_{S_4^-} \ni_i \{B_3, B_3^+\}_{S_4^-} \ni_i \{\{k\}_{A_2^-}\} \ni_i (i) \ni_i \{\{k\}_{A_2^-}B_3^+\}$$

The construction of the previous message let the trace end in the final state $\Sigma_{12}$:

$$\Sigma_{12} = \langle \{0\}, \emptyset \rangle, \langle \{0\}, \emptyset \rangle, \langle \{xa_2 \rightarrow S_1, y_0 \rightarrow I_0, xb_3 \rightarrow S_1, xs_1 \rightarrow A_2, ys_1 \rightarrow I_0, za_2 \rightarrow I_0^+, xs_4 \rightarrow I_0, ys_4 \rightarrow B_3, wb_3 \rightarrow A_2, zb_3 \rightarrow k \}, \{I_0, I_0^+, I_0^-, S_1^+, A_2, B_3, S_4, S_4^+, (A_2, I_0), (\{A_2, A_2^+\}_{S_4^+}, \{I_0, I_0^+\}_{S_4^+})\}, \langle \{A_2, A_2^+\}_{S_4^-}, \{I_0, I_0^+\}_{S_4^-}, \{\{k\}_{A_2^-}\}_{I_0^+}, (A_2, A_2^+ s_4^+, B_3, B_3^- s_4^-) \rangle \rangle$$

To show that the above trace is an attack (Definition 5.16), it rests to prove that $\kappa_{12} \not\models_{\chi_{12}} \phi$. We try to conclude the proof for $\kappa_{12} \models_{\chi_{12}} \phi$, showing that it is not the case.
As expected, the part of $\phi$ regarding the secrecy of $k$ can be satisfied, in fact $k$ cannot be derived from $\kappa_{12}$. Differently, to show that the proof cannot be concluded in $(\ast\ast\ast)$, the part regarding authentication, it is enough to show that both the rules that can be applied for $\lor$ do not permit the proof to be concluded:

$$\frac{\kappa_{12} \not\vdash \chi_{12} \land \forall i : B. (\lnot (\kappa \triangleright z b_i))}{\kappa_{12} \not\vdash \chi_{12} \land \forall i : B. (\lnot (\kappa \triangleright z b_i))} \quad (\ast)$$

which cannot be concluded, in fact

$$\frac{yb_3 \chi_{12} = A}{\kappa_{12} \vdash \chi_{12} \land \forall i : B. (\lnot (\kappa \triangleright z b_i))} \quad (=)$$

and hence the proof for $\lor 1$ does not conclude. The proof for $\lor 2$ does not conclude either:

$$\frac{ya_0 \chi_{12} \neq I_0}{\kappa_{12} \vdash \chi_{12} \land \forall i : B. (\lnot (\kappa \triangleright z b_i))} \quad (=)$$

$$\frac{\kappa_{12} \not\vdash \chi_{12} \land \forall i : B. (\lnot (\kappa \triangleright z b_i)) \lor ya_0 = B_3}{\kappa_{12} \not\vdash \chi_{12} \land \forall i : B. (\lnot (\kappa \triangleright z b_i)) \lor ya_0 = B_3} \quad (\lor 2)$$

The trace is hence an attack.

### 5.7 Symbolic protocol verification

Symbolic analysis is introduced to deal with the infinite branching of the state space, due to the capability of the intruder to generate an infinite number of different input messages for a principal instance. Infinite branching makes the previously proposed procedure not effective.

The basic idea of symbolic analysis is the following:

> Whenever a principal instance is ready to preform an input action whose data is a variable or contains a variable, the generation of a corresponding message from the intruder knowledge is delayed.
The finite representation of the whole set of messages that can be currently derived by the intruder, i.e. the current set $\kappa$, is associated to the input variable, which then becomes a “symbolic variable”.

The subsequent usage of the symbolic variable in the trace may further constrain the set of the values that the symbolic variable can assume, up until determining a unique possible message. The knowledge associated to the symbolic variable is correspondingly updated.

A symbolic trace corresponds to the non symbolic (concrete) traces, generally infinite in number, that can be obtained by suitably instantiating its symbolic variables. A soundness result guarantees that all the traces obtained by instantiating symbolic variables with concrete messages that can be derived from their associated knowledge, are traces according to the concrete semantics.

A new notion of formula satisfaction is necessary in order to verify formulas against final states of symbolic trace which may contain symbolic variables.

This is a lazy strategy, which delays the generation of a message so as to exploit the further information that may be later collected. By associating the current knowledge to the input variable no message is lost among those that could have been sent.\footnote{This paves the way to prove completeness results, which, for the sake of simplicity, are not presented here. Indeed, a detailed treatment of the symbolic approach is beyond the scope of this thesis.} Associating the knowledge $\kappa$ to a variable $xa$ yields the symbolic variable $xa(\kappa)$.

For instance, if a principal is ready to perform action $in(xa)$, the concrete semantics associates to $xa$ a message $m$ such that $\kappa \triangleright m$ (with $\kappa$ the current knowledge), while the symbolic semantics associates the symbolic variable $xa(\kappa)$, and correspondingly the substitution $xa \rightarrow xa(\kappa)$ is added to the current $\chi$.

It might be the case that the continuation of a trace further constrains the values that can be associated to a symbolic variable. It is the case, for instance, of a symbolic variable used as cryptographic key, like in, e.g. $in(\{?ya\}_{xa(\kappa)})$ ($xa(\kappa)$ has been received in a previous communication). In this case, the symbolic message is constrained to be one of the keys, if there are any, that can be generated from $\kappa$. A finite number of different messages, each one corresponding to a different key $\lambda \in \kappa$, can be generated in order to match the input action. For each of them the symbolic variable is correspondingly restricted ($xa(\kappa) \rightarrow \lambda$).

Restricting symbolic variables, as done before for the case of keys, is meant to guarantee a strict correspondence between a symbolic trace and the concrete one it represents. The case corresponding to keys, for instance, prevents messages different from keys being represented by a symbolic variable occurring in place of a key.
5.7.1 A symbolic extension of cIP-calculus

Adopting symbolic techniques requires a redefinition of the theory developed (Section 5.2, Section 5.3 and Section 5.4), starting from the inclusion of symbolic variables into the messages that can be communicated. For most of the theory — which we do not recall here — the extension to the symbolic case is straightforward. We, instead, define the concepts that are mostly influenced by the introduction of symbolic variables into the set of messages.

Definition 5.17 (Symbolic messages, data and knowledge) Let $\kappa$ be a knowledge, and $V(\kappa) = \{wa(\kappa), wb(\kappa), xa(\kappa), \ldots\}$ the set of symbolic variables. Then the set of symbolic messages $sM$ extends the set of messages $M$ as follows:

$$sM ::= PN | K | PN^+ | PN^- | NO | V(\kappa) | \{sM\}_K | \{sM\}_{PN^+} | \{sM\}_{PN^-} | (sM, sM).$$

A symbolic knowledge $\kappa$ is a finite subset of $sM$. Let $V$ be a set of variables, then the set of symbolic data $M_V$ is defined according to the following syntax:

$$sM_V ::= PN | K | PN^+ | PN^- | NO | V(\kappa) | V | ?V | \{sM\}_{sM} | (sM, sM).$$

Derivation of a symbolic message from a symbolic knowledge simply extends derivation of non-symbolic messages. After having proved that constructive derivation (Definition 5.3) is decidable (Theorem 5.3), we present constructive symbolic message derivation.

8

Definition 5.18 (Symbolic message derivation ($\triangleright$)) A symbolic message $m \in sM$ can be derived from a symbolic knowledge $\kappa$, if $\kappa \triangleright m$ can be proved by the following rules:

$$\frac{m \in \kappa}{\kappa \triangleright m}$$

$$\frac{\kappa \triangleright m}{\kappa \triangleright (m, n)}$$

$$\frac{\kappa \triangleright \lambda}{\kappa \triangleright \{m\}_\lambda}$$

$$\frac{\forall m \in \kappa'. \kappa \triangleright m}{x_i(\kappa')}$$

$i$

$s_i$

The rule $s_i$ states that a (symbolic) knowledge $\kappa$ can derive any symbolic variable whose associated (symbolic) knowledge $\kappa'$ contains a (finite) number of (symbolic) messages that can be (symbolically) derived from $\kappa$.

The indexing of messages, data, symbolic and non-symbolic variables and principals, as well as variable binding, work as in the non-symbolic case. In the following, indexes may sometimes be omitted for the sake of simplicity.

Some care is necessary to extend substitution to the symbolic case. Indeed, symbolic variables, which play the part of special messages, can also be substituted with either non-symbolic or symbolic messages. According to the interpretation of

8For the ease of presentation, we do not redefine all the theory, but rather abuse the notation, so that, for example, $\kappa$ is now a symbolic knowledge. This ambiguity is resolved by the context in which terms are used.
symbolic variables as annotations of the knowledge owned by the intruder at a given instant, it is required that, when substituting symbolic variables, the associated knowledge is not enlarged.

Composition of substitutions works as in the non-symbolic case, considering each symbolic variable as a normal variable identified by its name (e.g. if $I \in sM_i$ composing \([x_i \rightarrow x_i(\kappa)]\) and \([x_i(\kappa) \rightarrow I]\), yields \([x_i \rightarrow I]\) — provided that $\kappa \triangleright I$).

**Definition 5.19 (Symbolic and concretising substitution)** A symbolic substitution is a mapping $\sigma: V \cup V(\kappa) \rightarrow sM$, from symbolic variables and variables to symbolic messages, such that for each symbolic variable $x(\kappa)$ on which it is defined, it holds

$$\kappa \triangleright \sigma(x(\kappa)).$$

A concretising substitution is a symbolic substitution $\psi: V(\kappa) \rightarrow M$ whose range is contained in the set of non-symbolic messages $M$.

Composition of substitution is indicated as “;”, and $\epsilon$ is the empty substitution.

Symbolic semantics requires an appropriate definition of symbolic matching, $\sim_s$, extending the concept of matching data to the case of symbolic variables.

**Definition 5.20 (Symbolic matching)** Let $m, n \in sM$, then $m$ and $n$ symbolically match if $m \sim_s n$ holds. Where

$$\begin{aligned} m \sim_s n \iff \quad & \quad m = x_i(\kappa_s) \land n = x_i(\kappa_s) \\
& \quad m = (e, f) \land n = (g, h) \land e \sim_s g \land f \sim_s h \\
& \quad m = \{e\}_{\lambda} \land n = \{g\}_{\lambda'} \land e \sim_s g \\
& \quad m = e \land n = e \land e \in PN_i \cup NO_i \cup K_i \cup PN_i^+ \cup PN_i^- 
\end{aligned}$$

Matching two symbolic variables in a communication means that the two must correspond to the same set of possible messages, i.e. they must be the same symbolic variable, i.e. a symbolic variable can be matched only by itself. Moreover, some care is necessary to deal with asymmetric cryptograms. The two actions \(\text{in}(x_i(\kappa))\) and \(\text{out}(x_i(\kappa))\) synchronise if their data match. In this case the data are actually the same symbolic variable, and one would like to guarantee that anyone of the messages derivable from $\kappa$ can be used to let the two actions synchronise. The problem, as usual, arises with asymmetric cryptograms, that do not match with themselves. Let us suppose that $\kappa = \{\text{no}_1, A_i^+\}$, while the (concrete) choice of $A_i^+$ for $x_i$ is correct, since $A_i^+ \sim A_i^+$, the message $\{\text{no}_1\}_{A_i^+}$ should not be derived from $\kappa$, since $\{\text{no}_1\}_{A_i^+} \not\sim \{\text{no}_1\}_{A_i^-}$ (but rather $\{\text{no}_1\}_{A_i^+} \sim \{\text{no}_1\}_{A_i^-}$).
E_i^{\text{out}(m')} \rightarrow E'_i \quad \exists \sigma : m'\sigma \in sM \quad (I - \text{in})

\langle (X_i)[E_i] \cup C, \chi, \kappa \rangle \xrightarrow{(m'\sigma)} \langle (X_i)[E'_i\sigma] \cup \{ m'_\sigma \} \rangle

E_i^{\text{in}(m')} \rightarrow E'_i \quad \exists \sigma, m : \kappa \triangleright m \land m\sigma \sim_s m'_\sigma \quad (I - \text{out})

\langle (X_i)[E_i] \cup C, \chi, \kappa \rangle \xrightarrow{(m')} \langle (X_i)[E'_i\sigma] \cup \{ m'_\sigma \} \rangle

\mathcal{C}' = \text{join}(A_i, \gamma, \mathcal{C}) \quad A_i = (X_i)[E_i] \quad \text{i new} \quad (I - \text{join})

\langle \mathcal{C}, \chi, \kappa \rangle \xrightarrow{\text{join}(A_i, \gamma)} \langle \mathcal{C}', \chi, \gamma, \kappa \sqcup A_i \rangle

Figure 5.4: Symbolic session semantics.

The introduction of the annotation $\ast$ for $\kappa_\ast$ in the symbolic matching, has been necessary in order to solve this problem: no asymmetric cryptograms can be derived from $\kappa_\ast$. This is obtained by overriding the rules for $\triangleright$.\footnote{Note that simply discharging asymmetric keys form $\kappa$, could result in loosing the concrete traces in which asymmetric keys are not used to encrypt a message.}

The other cases of the definition reduce to the corresponding cases in the definition of the non-symbolic matching. Other possible cases, e.g. a symbolic variable $x_i(\kappa)$ and a non-symbolic message $m$, will be dealt with by means of an appropriate substitution, if it exists, such that $x_i(\kappa)\sigma \sim_s m\sigma$ according to the previous definition.

The symbolic semantics of Figure 5.4 corresponds to the concrete one of Figure 5.3, but for the presence of symbolic messages. Principal instances in $\mathcal{C}$ may contain symbolic messages, as well as the knowledge $\kappa \subseteq sM$ can, and $\chi : X_i(\kappa) \cup X \rightarrow sM$ is a symbolic substitution. (The principal semantics $\rightarrow$ (Figure 5.2) is, straightforwardly, extended to the symbolic case). Differently from concrete semantics:

- Rule $(I - \text{in})$ requires that each message sent by a principal instance is a proper symbolic message in which symbolic variables do not occur as keys. In the case that a symbolic variable $xa_i(\kappa)$ occurs as a key, the transition can happen if a symbolic substitution exists that maps the variable in one of the keys derivable from $\kappa$. The possible non deterministic choices are finite in number, and do not generate infinite branching. All the occurrences $xa_i(\kappa)$ appearing in the session, in $\chi$ and in $\kappa$ are accordingly updated by the application of $\sigma$. The knowledge is also enlarged with the sent symbolic message $m'\sigma$.

- In the case that a principal instance is ready to input a message $m'$, rule $(I - \text{out})$ requires the existence of a message $m$, which can be derived from
5.7. SYMBOLIC PROTOCOL VERIFICATION

\( \kappa \), and of a substitution \( \sigma \) such that \( m\sigma \) and \( m'\sigma \) symbolically match. The substitution \( \sigma \) can, as usual, instantiate the variables in \( m' \) and also restrict the symbolic variables occurring in both the messages. It is applied, as in the previous point, to the session, to \( \chi \) and to \( \kappa \).

- Rule \((I - join)\) works as in the non symbolic semantics, in fact, a joining instance does not contain symbolic messages.

Differently from the case of concrete semantics, producing, if they exist, the substitutions and messages required by the existential premises of rules \((I - in)\) and \((I - out)\) consists of a terminating algorithmic procedure. It is worth pointing out that, given a message \( m' \) of any form, the number of corresponding symbolic messages and substitutions generated is finite. Section 5.7.3 illustrates the procedure.

Symbolic traces work as expected.

**Definition 5.21 (Symbolic trace)** A state \( \langle \{\}, \chi, \kappa \rangle \) is initial if and only if \( \chi \) is the empty substitution and \( \kappa \) is a finite subset of \( M \). It is final if and only if \( \chi \) is a symbolic substitution and \( \kappa \) is a finite subset of \( sM \). A symbolic trace is a sequence

\[
T_s = \Sigma_0.\alpha_1.\ldots.\alpha_n.\Sigma_n
\]

where \( \Sigma_0 \) is an initial state and \( \Sigma_{i-1} \leadsto \Sigma_i \) \((1 \leq i \leq n)\). \( T_s \) is terminating if \( \Sigma_n \) is a final state.

5.7.2 Correctness of symbolic protocol verification

The practical use of symbolic verification relies on a correspondence between the traces of symbolic semantics and the traces of concrete semantics. As a complete treatment of the symbolic approach is beyond the scope of this thesis, we give a correctness theorem that states the soundness of symbolic semantics with respect to the concrete one. This theorem allows us to verify symbolically, and hence effectively, protocols, in the sense that, given a (final) symbolic state, all the concrete states obtained by applying to it a concretising substitution, can actually be reached by a concrete trace. Hence, a symbolic trace, which is interesting because it makes a given property hold, can effectively be used to generate a corresponding concrete trace, where the property still holds. In a word, symbolic attacks constructively correspond to concrete attacks. The following theorem states the correspondence between symbolic and concrete traces.

**Theorem 5.4 (Soundness of symbolic semantics)** Let \( \Sigma_0 \leadsto^n \Sigma_n \) be a symbolic trace such that \( \Sigma_n = \langle \mathcal{C}, \chi, \kappa \rangle \), then it holds

\[
\forall \psi \exists \Sigma_0 \leadsto^n \Sigma_n^c \text{ and } \Sigma_n^c = \langle \mathcal{C} \psi, \chi \psi, \kappa \psi \rangle,
\]

where \( \Sigma_0 \leadsto^n \Sigma_n^c \) is a (concrete) trace and \( \psi \) a concretising substitution such that \( \chi \psi : V \cup V(\kappa) \rightarrow M \).
Proof. See Section 5.10

The theorem relies on the capability of suitably generating symbolic messages. Intuitively speaking, the capability of generating only “significant” symbolic messages leads to the above theorem, while the capability of generating all the “significant” symbolic messages is a requirement for proving completeness results.

5.7.3 Generating symbolic messages

Symbolic semantics (Section 5.7.1) is based on the existence of $\sigma, m$ such that, given a symbolic message $m'$:

$$\exists \sigma : m' \sigma \in sM \quad (I - in)$$

and

$$\exists \sigma, m : \kappa \triangleright m \land m \sigma \sim_s m' \sigma \quad (I - out).$$

This section explains how $\sigma$ and $m$ can be constructively determined.

Input messages

A principal instance may, at a given point in a trace, be ready to output a symbolic message, which may possibly contain symbolic variables, generated by the intruder and received in previous communications. While, in the concrete semantics, the erroneous use of a non-key in place of a key leads to a run-time error, the case of a symbolic variable occurring as a key corresponds to all (and only those) concrete messages, and hence transitions, in which the symbolic variable is actually constrained to be a key, if there is one, chosen among those that can be derived from the associated knowledge, (see symbolic semantics, Figure 5.4). Note that the number of such keys is finite (this would not be true if non-atomic keys were allowed).

For instance, a principal behaving as $\text{in}(\text{xa}_1).\text{out}((\text{xa}_1)_{\text{xa}_1})$, which receives the symbolic variable $\text{xa}_1(\kappa)$ in its first action, will be able to output the message $(\text{xa}_1(\kappa))_{\text{xa}_1(\kappa)}$, if and only if $\text{xa}_1(\kappa)$ stands for one of the keys, if there is one, which can be derived from $\kappa$.

The following function $\ell$ takes a symbolic datum $m \in sM_V$ and returns a substitution $\sigma$ such that $m \sigma \in sM$ is a symbolic message (without non-keys occurring as keys). The function is defined by structural induction on $m$ and is non-deterministic, returning one of the possible, finite in number, substitutions. This generates a finite set of possible transitions, one for each possible substitution. Non-determinism, is traditionally expressed by the construct $\sigma = \text{choice}(\theta)$, which either assigns to $\sigma$ one of the elements in the set $\theta$, or fails if $\theta = \emptyset$. The function $\ell$ is hence partial, e.g. it fails on the message $(\text{no}_1)_{\text{x}_1((\text{no}_2))}$, and failure is represented as $\bot$. 
The definition of $\ell$ is straightforward. It is worth pointing out that, thanks to the non-deterministic definition of $\ell$, all the possible choices for the symbolic variables which correspond to a concrete transition are generated.

**Definition 5.22 ($\ell$)** Let $\epsilon$ be the empty substitution. Then, the non-deterministic partial function $\ell : sM_V \rightarrow ((V_i(\kappa) \rightarrow PN_i^+ \cup PN_i^- \cup K) \cup \bot)$ is defined as follows:

$$
\ell(m) = \begin{cases} 
\epsilon & m \in PN_i \cup PN_i^+ \cup PN_i^- \cup K \cup NO_i \cup V_i() \\
\ell(n) & m = \{n\}_{\lambda} \\
\ell(n\sigma); \sigma & m = (n, o) \land \\
& \sigma = (\ell(o)) \\
\ell(n\sigma); \sigma & m = \{n\}_{x_i(\kappa)} \land \\
& \lambda = \text{choice} (\{\lambda' | \lambda' \in \kappa\}) \land \\
& \sigma = x_i(\kappa) \mapsto \lambda \\
\bot & \text{otherwise}
\end{cases}
$$

**Output messages**

Given an input message $n$ expected by a principal and a knowledge $\kappa$, the intruder output message $m$ and a substitution $\sigma$, such that $\kappa \triangleright m$ and $m\sigma \sim_s n\sigma$, are constructed by the function $\mu(n, \kappa)$, defined by structural induction on $n$. Such function is non-deterministic and returns one of possible, finite in number, couples $(m, \sigma)$ satisfying the requirements. We assume that $n \in sM$ is a symbolic message, i.e. it does not contain symbolic variables occurring as keys.\(^\text{10}\)

**Definition 5.23 ($\mu$)** Let be $n \in sM_V$ and $\kappa \subset sM_i$. The partial function $\mu : (sM_V \times 2^{sM_i}) \rightarrow ((sM_i \times ((V_i(\kappa) \cup V_i) \rightarrow sM_i)) \cup \bot)$ is defined as follows:

\(^{10}\)The assumption, other than simplifying the presentation, is reasonable. Otherwise, the $\mu$ function would necessarily be in charge of resolving symbolic variables as keys in order to allow the communication between the intruder and the principal instance. Differently we consider that these cases are resolved beforehand, i.e. the function is always called as $\mu(\ell(n), \kappa)$. It is understood that the concretising substitution $\sigma = \ell(n)$ must be composed with the substitution returned by $\mu(\ell(n), \kappa)$. 

\[
\mu(n, \kappa) = \begin{cases}
(x_i, x_i \mapsto x_i(\kappa)) & n = ?x_i \\
(n, \epsilon) & n \in NO_i \cup K \cup PN^+_i \cup PN^-_i \cup PN \land \kappa \triangleright n \\
(x_i(\kappa'), x_i(\kappa') \mapsto x_i(\kappa')) & n = x_i(\kappa') \\
((e', f'), \sigma_e; \sigma_f) & n = (e, f) \land \mu(e, \kappa) = (e', \sigma_e) \land \\
& \mu(f \sigma_e, \kappa \sigma_e) = (f', \sigma_f) \\
\{e'\}_\lambda, \sigma_e) & n = \{e\}_\lambda \land \kappa \triangleright \lambda \land \\
& \mu(e, \kappa) = (e', \sigma_e) \\
\{e'\}_\lambda, \sigma_e) & n = \{e\}_\lambda \land \\
& \exists \{e'\}_\lambda = \text{choice}(\{\{f\}_\lambda | \{f\}_\lambda \in \kappa\}) \land \\
& \exists \sigma_e, \nu(e', e) = \sigma_e \\
\bot & \text{otherwise}
\end{cases}
\]

If \(n\) is a binding variable \(?x_i\), then the output message \(m\) is the symbolic variable relative to the current \(\kappa, x_i(\kappa)\), and the substitution is trivially \(x_i \mapsto x_i(\kappa)\).

If \(n\) is an atomic non-symbolic message, that can be derived from \(\kappa\), then \(m = n\) and the substitution is obviously empty.

If \(n\) is a symbolic variable \(x_i(\kappa')\), then it must have been previously generated by the intruder, and, since the knowledge monotonically increases through a trace, it can obviously be derived from the current \(\kappa\). It is however necessary, according to Definition 5.20, to restrict \(\kappa'\) to \(\kappa_s\).

If \(n = (e, f)\) then \(\mu\) is firstly applied to the sub-message \(e\). If this (non-deterministically) returns a result \((e', \sigma_e)\), then it is propagated to the call for \(f\): \(\mu(f \sigma_e, \kappa \sigma_e)\). If this latter application non-deterministically returns a value \((f', \sigma_f)\), then the overall returned value is \(((e', f'), \sigma_e; \sigma_f)\).\(^{11}\)

If the input message \(n\) is a cryptogram \{e\}_\lambda, and \(\lambda\) can be derived from \(\kappa\), then the problem reverts to checking if it is possible to derive a message \(m'\) matching \(e\) (and a corresponding substitution), since then, the required cryptogram \{m'\}_\lambda can be derived.

On the other hand, it might also be the case that one or more “potentially matching” cryptograms \{e'\}_\lambda belong to \(\kappa\), independently of whether the key \(\lambda\) belongs or

\(^{11}\)Note that the order in which the couple \((e, f)\) is visited is not relevant. Function \(\mu\) is lazy in the sense that it fixes values only if they are necessary. If a choice for \(e\) results incompatible with the subsequent choices for \(f\), it would not result compatible only because the choices for \(f\) have been done earlier. Moreover, the non-deterministic nature of the function guarantees that all the possible choices for \(\sigma_e\) and \(\sigma_f\) are taken into consideration, and hence \(\sigma_e; \sigma_f\) is chosen among all the possible combinations of the choices for \(\sigma_e\) and \(\sigma_f\).
5.7. SYMBOLIC PROTOCOL VERIFICATION

not to $\kappa$. Note that, in absence of the key $\lambda$ (resp. $\lambda^-$), such cryptograms can not be “constructed” (resp. “de-structed”) from the remaining of $\kappa$. It is anyway possible to constrain their symbolic variables (which are determined by the intruder) in order to let them match $n = \{e\}_{\lambda^+}$. In other words, it is only possible to check for the existence of a substitution $\sigma$, that makes $\{e\}_{\lambda}$ and $n = \{e\}_{\lambda^-}$ match each other (note that $\sigma$ maybe instantiate variables in $n$, too). This is done by the auxiliary function $\nu$, defined below ($\nu(m, n) = \sigma \iff m\sigma \sim_s n\sigma$).

If none of the previous cases applies, the function $\mu$ fails (i.e. $\perp$). Note that all the cases of non-determinism for $\ell$ and $\mu$ relies on a choice over finite sets.

**Definition 5.24 ($\nu$)** Let be $m, n \in sM_{Vi}$, and $\kappa \subset sM_i$. The partial function $\nu : (sM_{Vi} \times sM_{Vi}) \rightarrow (((V_i(\kappa) \cup V_i) \rightarrow sM_i) \cup \perp)$ is defined as follows:

$$
\nu(m, n) = \begin{cases} 
  x_i \mapsto m & n = x_i \quad (m \notin V_i) \\
  e & n = m \in NO_i \cup K \cup PN_i^+ \cup PN_i^- \cup PN_i \\
  \nu(e, e') & m = \{e\}_\lambda \land n = \{e'\}_\lambda^- \\
  \sigma_e; \sigma_f & m = (e, f) \land n = (e', f') \land \\
  \nu(e, e') = \sigma_e \land \nu(f, f') = \sigma_f \\
  y_i(\kappa') \mapsto m'; \sigma & m \notin V_i(\kappa) \land n = y_i(\kappa') \land \\
  \mu(\kappa', m) = (m', \sigma) \\
  x_i(\kappa') \mapsto n'; \sigma & m = x_i(\kappa') \land n \notin V_i(\kappa) \land \\
  \mu(\kappa', n) = (n', \sigma) \\
  x_i(\kappa') \mapsto x_i((\kappa \cap \kappa')_s); & m = x_i(\kappa) \land n = y_j(\kappa') \land \\
  \kappa \cap \kappa' = \{m \in \kappa \cup \kappa' \mid m \land \kappa' \lor m \} \neq \emptyset \\
  y_j(\kappa') \mapsto x_i((\kappa \cap \kappa')_s); & m = x_i(\kappa) \land n = y_j(\kappa') \land \\
  \kappa \cap \kappa' = \{m \in \kappa \cup \kappa' \mid m \land \kappa' \lor m \} \neq \emptyset \\
  \perp & otherwise
\end{cases}
$$

The function $\nu$, of Definition 5.24, given the two symbolic data $m, n$, returns, if there is one, a unifying substitution $\sigma$ such that $m\sigma \sim_s n\sigma$. Note that the function $\mu$ and $\nu$ are mutually recursive: The function $\mu$ calls the function $\nu$ to verify if $\{e\}_\lambda$ and $\{e'\}_{\lambda^-}$ may match under an appropriate substitution. The function $\nu$, instead, calls $\mu$ to verify whether a symbolic variable (occurring in $\{e\}_\lambda$) may generate a message matching a given message (occurring in $\{e'\}_{\lambda^-}$).

The first three cases are straightforward. For the fourth, the same considerations about the independence from the visiting order of the couple in $\mu$, apply.

\footnote{Whenever $\nu$ is used within $\mu$, it holds that $m \notin V_i$, as we assume in the definition of $\nu$.}
The cases in which $n$ (or $m$) is a symbolic variable $y_j(x_i(k'))$, but $m$ ($n$) is not, namely fifth and sixth cases, require us to check whether $k'$ can generate a message that matches $m$ ($n$). This is obviously done by calling $\mu(k', m)$ ($\mu(k', n)$), which by definition solves the problem. Note that at each recursive call of $\mu$ and $\nu$, the complexity of the argument decreases, so that their mutual recursion can not cause infinite loops. Indeed, the structural complexity of the argument decreases at each recursive call of any of both the two functions, which hence eventually terminate.

If both $m$ and $n$ are (possibly different) symbolic variables, last but one case, then the two can symbolically match only if they represent the same set of messages. This can be achieved by

1. unifying the two symbolic variables, by mapping them in the same variable (the one of $m$, say),

2. restricting the knowledge associated to the set of the messages which belongs to the intersection of the messages that the two symbolic variables can originally generate, and verify that it is not empty.

The set $\kappa \cap \kappa'$ is the maximal set of messages that can be derived from $\kappa$ and $\kappa'$, so that none of the possible concrete choices is lost. In general, the intersection of the derivable messages from two knowledges is not a function of the intersection of the knowledges. Consider, for example, $\kappa = \{\{no_1\}_k\}$ and $\kappa' = \{no_1, k\}$: The intersection is empty, but both the knowledges derive $\{no_1\}_k$. In this case $\kappa \cap \kappa' = \{\{no_1\}_k\}$.

Given a (symbolic) message $m$, it can be easily proved by structural induction on the proof trees that

$$\kappa \triangleright m \land \kappa' \triangleright m \iff \kappa \cap \kappa' \triangleright m.$$  

3. labeling the union knowledge with $\ast$, so as to let it match itself, (see Definition 5.20),

as is done by the corresponding case in the definition of $\nu$.

All the other cases correspond to messages that can not be unified.

Straightforwardly, since they are recursively invoked on terms with a decreasing structural complexity, and the possible non-deterministic choices are made on finite sets, the functions $\ell(m), \mu(\kappa, m)$ and $\nu(m, n)$ always terminate, non-deterministically producing a finite set of results.

**Example 5.10** The following examples illustrate how an appropriate message $m$, matching an input message $n$, may be derived from $\kappa$, where

$$\kappa = \{k, \{no_1\}_B, no_1, \{j_3(\{A^+_3, A^+_3\}_k)\}\}.$$
1. \( n = z_5(\{\{no_1\}_k,\{no_1\}^-_{B_2}\}) \),

is a symbolic variable, whose associate knowledge \( \kappa' \) contains the two messages: \( \{no_1\}_k \) and \( \{no_1\}^-_{B_2} \). We verify that \( z_5(\kappa') \) can be derived by the current \( \kappa \), even if this is assumed in the definition of \( \mu \) because of the monotony of \( \kappa \) throughout a trace.

The first message can be derived in one step, while the second one directly belongs to \( \kappa \), hence, by applying the last rule for \( \triangleright \), \( \kappa \triangleright z_5(\kappa') \). Hence, \( \mu(n,\kappa) = (n,z_5(\kappa') \mapsto z_5(\kappa'_s)) \).

2. \( n = \{\{?h_1\}^-_{B_2}\} x_2(\{k,A_3^-\},no_1) \),

is a symbolic datum which contains symbolic variables as keys. The first step of \( \mu(n,\kappa) \) is then to apply \( \ell(n) \), which non-deterministically yields the two substitutions \( \sigma_1 = x_2(\kappa') \mapsto k \) and \( \sigma_2 = x_2(\kappa') \mapsto A_3^- \), since \( no_1 \) can not occur as a key.

It is easy to verify that \( \mu(\{\{?h_1\}^-_{B_2}\} A_3^-,\kappa\sigma_2) = \bot \), since neither \( \kappa \) contains the key \( A_3^+ \), nor a cryptogram encrypted with it.

The case for \( \mu(\{\{?h_1\}^-_{B_2}\} A_3^-,\kappa\sigma_1) \) reverts to \( \mu(\{\{?h_1\}^-_{B_2}\} \kappa\sigma_1) \), since \( \kappa\sigma_1 \triangleright k \). From \( \kappa \triangleright \{no_1\}^-_{B_2} \) and \( \nu(no_1,?h_1) = h_1 \mapsto no_1 = \sigma \), it follows \( \mu(n,\kappa) = (\{\{no_1\}^-_{B_2}\} k,x_2(\kappa') \mapsto k;h_1 \mapsto no_1) \). Note that, indeed, \( \kappa \triangleright \{\{no_1\}^-_{B_2}\} k \) and \( \{\{no_1\}^-_{B_2}\} k\sigma_1; \sigma \sim \kappa \), \( \kappa \).

3. \( n = \{w_2(\{k,A_3^+\})\}^-_{C_4} \),

is a cryptogram containing a symbolic variable. The knowledge \( \kappa \) contains the cryptogram \( \{j_3(\{A_3^+\},\{A_3^-\}_k)\} A_3^- \), so that \( \mu(n,\kappa) \) 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 \( \kappa \cap \kappa' = \{A_3^+\} \), which does not contains \( k \), since \( \{A_3^+\},\{A_3^-\}_k \neq k \).

For example, \( (\kappa \cap \kappa') \triangleright \{A_3^+\}_k \), but not, for example, \( (\kappa \cap \kappa') \triangleright \{A_3^+\}_A_3^- \), since \( \{A_3^+\} \) cannot match itself.

The \( \nu \) function returns the following substitution:

\[
  w_2(\kappa \cap \kappa') \mapsto w_2((\kappa \cap \kappa')_s); j_3(\kappa) \mapsto w_2((\kappa \cap \kappa')_s),
\]

and hence \( \mu(n,\kappa) = (\{j_3(\{A_3^+\},\{A_3^-\}_k)\} A_3^- \), \( w_2(\kappa \cap \kappa') \mapsto w_2((\kappa \cap \kappa')_s)). \) It is again easy to check that \( \kappa \triangleright m \) and \( m\sigma \sim \kappa \).

Note that for all the above cases, any possible concretising substitution \( \kappa \) for \( m \) and \( n \), leads to concretely matching messages \( mn \psi \sim n\psi \).
The definition of the input/output message generation functions aims at guaranteeing the desired correspondence between concrete and symbolic semantics, which can be interpreted as

1. all the concretisations of a symbolic trace are correct concrete traces, and

2. given a concrete trace, there is a symbolic trace, such that the concrete one can be obtained by concretising it.

Theorem 5.4 proves 1., making symbolically reasoning about protocols safe. On the other hand, about completeness that is not dealt with here in detail, it is interesting to point out how unnecessarily restricting the values that a symbolic variable can assume causes symbolic semantics not to “represent” the concrete traces in which those values are communicated (just as improperly extending the knowledge associated to a symbolic variable, may cause the symbolic semantics to represent not existing concrete traces).

From the above examples, and from inspecting the definition of the functions ℓ and μ (and hence ν), it can be understood that the messages and the substitutions they return are as abstract as possible: similarly to the construction of a most general unifier, they add all and only the constraints which are necessary to let the symbolic session properly evolve via a transition, preserving the correspondence with the concrete semantics. In order to prove soundness, it can be easily proved that if μ(n, κ) returns a couple (m, σ) then κ ∪ m and mσ ∼ s nσ, and for any concretising substitution ψ, such that mσψ, nσψ ∈ M, then κσψ ∪ mσψ and mσψ ∼ nσψ. On the other hand, intuitively speaking, for completeness, a corresponding result would be: if m′ and σ′ exist, such that κ ∪ m′ and nσ′ ∼ s m′σ′, then μ(n, κ) returns a couple (m, σ), such that m is “more general” then m′ (i.e. ∃σ′′. mσ′′ = m′).

5.7.4 Symbolic trace generation: An example

Symbolic analysis is applied to a simple protocol by showing the generation of a symbolic trace, constituting a candidate attack. Section 5.7.6 shows the formulation of a security property in terms of PL logic, and its verification with respect to the trace here generated. Among the finitely many traces that belong to the symbolic semantics, we have chosen the most suitable one to illustrate trace construction and property verification.

The protocol consists of three principals: A server S sends A and B a session key k encrypted with their respective public keys. B uses the session key to send A message no. One would (naively) expect that the communication between A and B is safe.

Open variables of S are intended for the the public keys of A and B, respectively:

\[
\begin{align*}
S \rightarrow A & : \{k\}_{A^+} & S & \triangleq (xs, ys)[\text{out}(\{k\}_{xs^+}).\text{out}(\{k\}_{ys^+})] \\
S \rightarrow B & : \{k\}_{B^+} & A & \triangleq ()[\text{in}(\{?ua\}_{A^-}).\text{in}(\{?wa\}_{ua})] \\
B \rightarrow A & : \{no\}_{k} & B & \triangleq ()[\text{in}(\{?vb\}_{B^-}).\text{out}(\{no\}_{vb})]
\end{align*}
\]
Initially, the intruder knowledge contains the symmetric keys of the intruder itself, named $I_0$, a message $(mm_0)$ and a key $(kk)$ different from those appearing in the protocol, $\kappa_0 = \{I_0, I_0^-, mm_0, kk\}$. This is a reasonable assumption, corresponding to the capability of the intruder of generating new names. The initial state is:

$$\Sigma_0 = \langle\{\}, \varepsilon, \{I_0, I_0^-, mm_0, kk\}\rangle$$

Among the possible choices, the trace we are generating starts with joining the instances of all the three principals to the session.

$$\Sigma_0 \xrightarrow{j(S_1;\varepsilon)} j(A_2;x_1\mapsto A_2) \xrightarrow{j(B_3;y_1\mapsto B_3)} \Sigma_3 = \langle\{([\text{out}(\{k\}A^+_3).\text{out}(\{k\}B^+_3)]
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\q
Let us choose the trace relative to \(vb_3(\kappa_5) \mapsto kk\) (other possible choices):

\[
\Sigma_7 \overset{i(\{no\}_{kk})}{\sim} \Sigma_8 = \langle \{ \cdot \} \in \{ wa_2(\kappa_5) \} \rangle,
\]

with \(\chi_8 = \{ x_1 \mapsto A_2, y_1 \mapsto B_3, u_2 \mapsto ua_2(\kappa_5), v_3 \mapsto kk \}\) and \(\kappa_8 = \kappa_7 \cup \{ no \}_{kk}\). The last step requires again the concretization of a symbolic variable used as a key. Depending on the concretization, the intruder could either send an encryption which already belongs to its current knowledge, or a newly encrypted message. We show the case in which \(kk\) is used as a key and \(wa_2\) is hence associated to \(wa_2(\kappa_8)\):

\[
\Sigma_8 \overset{o(\{wa_2(\kappa_8)\}_{kk})}{\sim} \Sigma_9 = \langle \{ \cdot \} \in \{ wa_2(\kappa_8) \} \rangle,
\]

with \(\chi_9 = \{ x_1 \mapsto A_2, y_1 \mapsto B_3, u_2 \mapsto kk, v_3 \mapsto kk, wa_2 \mapsto wa_2(\kappa_8) \}\) and \(\kappa_9 = \{ I_0, I_0^o, mm_0, kk, S_1, A_2, B_3, \{ k \}_{A_2^+}, \{ k \}_{B_3^+}, \{ no \}_{kk}, no_3 \}\) in explicit form (Definition 5.4).

The intruder grabbed the key that \(S_1\) has sent to \(A_2\) and \(B_3\), it sent them a different key \(kk\), and it has been able to modify the supposed safe communication between \(A_2\) and \(B_3\).

### 5.7.5 Symbolic verification

The syntax of the logic formulas, and the way property are formalised, is not influenced by symbolic analysis. However, when automatically checking if a symbolic trace generates a model for a formula, the \(\chi\) and the knowledge \(\kappa\), to which \(\chi\) has been applied throughout the trace, may contain symbolic variables, and hence a new notion of model is necessary.

For instance, a formula, instantiated by a symbolic substitution \(\chi\), holds depending on the set of values that can be or can not be derived from its symbolic variables. For example, satisfaction of \(xa_1[xa_1 \rightarrow xa_1(\kappa)] = B_3\) depends on whether \(\kappa \triangleright B_3\) holds.

We define a symbolic notion of model in terms of the concrete one, via a concretising substitution, and then provide an algorithmic procedure which produces a concretising substitution, if there is one, which makes the formula hold.

**Definition 5.25 (Symbolic model)** Let \(\chi : V \cup V(\kappa) \rightarrow sM\) be a symbolic substitution, \(\psi\) a concretising substitution, and \(\phi\) a formula of \(PL\). Then \(\kappa \subset sM\) is a...
5.7. SYMBOLIC PROTOCOL VERIFICATION

symbolic model for the formula \( \phi \) under to the substitution \( \chi \), written \( \kappa \models^* \chi \phi \), if and only if \( \exists \psi. \kappa \psi \models_{\chi \psi} \phi \).

In the previous, easy, case, the concretising substitution which makes the formula hold is trivially \( x\alpha_1(\kappa) \rightarrow B_3 \). It is worth remembering that a concretising substitution, by definition, maps symbolic variables into concrete messages that can be derived from the knowledge associated to the symbolic variables. According to Theorem 5.4, if \( \chi \) is the outcome of a symbolic trace, then \( \chi \psi \) is hence the outcome of a concrete trace, for all the possible concretising substitutions \( \psi \).

Hence, a symbolic model for a formula exists, (if and) only if a concrete model, which is a concretisation of it, exists for the same formula.

Definition 5.25 is not constructive, but the procedure for constructively checking for the existence of such a \( \psi \) is easy.

Algorithmic verification for “symbolic” formulas

Given a formula \( \phi \), we illustrate a procedure to algorithmically define a concretising substitution \( \psi \), if there is one, such that \( \kappa \psi \models_{\chi \psi} \phi \), where \( \chi \) and \( \kappa \) result from a terminating symbolic trace. Section 5.7.6 applies the procedure to a simple protocol.

Let us consider the formula \( \phi \) that implies the unsafeness of a given protocol. The process of verifying its satisfiability, with respect to a symbolic trace represented by \( \chi \) and \( \kappa \), consists of three steps:

1. **formula normalisation** — \( \phi \) is translated in an equivalent formula \( \phi' \) consisting in a disjunction of conjuncts, from which quantifiers have been eliminated according to the symbolic trace generating \( \kappa \) and \( \chi \). The transformation consists of:

   - **disjunction of conjuncts** — \( \phi \) is translated, by a standard truth preserving transformation, in an equivalent formula consisting of a disjunction of conjuncts of literals. A literal can be equality =, inequality \( \neq \), membership \( \triangleright \), non-membership \( \not\triangleright \). For example,
     
     \[
     \forall j : A \exists i : B. \neg (no_j \triangleright \kappa \land (xb_i = A_j \land yb_i = no_j))
     \]

     becomes

     \[
     \forall j : A \exists i : B (no_j \not\triangleright \kappa \land xb_i \neq A_j) \lor (no_j \not\triangleright \kappa \land yb_i \neq no_j).
     \]

   - **eliminating quantifiers** — Universal quantifiers, \( \forall i : A \), are replaced by \( h \) conjunctions if there are \( h \) instance indexes \( n_h \) such that \( A_{n_h} \in \kappa \). Similarly, existential quantifiers, \( \exists i : B \), are replaced by disjunctions. For example, \( \exists i : no_i \triangleright \kappa \) is substituted by \( no_3 \triangleright \kappa \lor no_5 \triangleright \kappa \lor no_7 \triangleright \kappa \) if 3,5,7 are the indexes of the instances of the principal in which \( no \) occurs and that have joined the session.
2. **actualisation with respect to a terminating symbolic trace** — the substitution \( \chi \) is applied to \( \phi' \) yielding \( \phi'' \), which may contain symbolic variables.

3. **validation of the formula** — \( \phi'' \equiv (\phi_1^{(1)} \land \cdots \land \phi_{n_1}^{(1)}) \lor \cdots \lor (\phi_1^{(h)} \land \cdots \land \phi_{n_h}^{(h)}) \) holds if any of its disjuncts \((\phi_1^{(j)} \land \cdots \land \phi_{n_j}^{(j)})\) holds.

Satisfiability of each disjunct depends on \( \triangleright \) atomic formulas, equalities and inequalities of variables ranging over an infinite domain (the set of messages). This represents a known (and solved) problems in constraint literature: equalities (that can be generated by \( = \) and \( \triangleright \) formulas) are dealt with by unification, then inequalities (over infinite domains) always have a solution, unless in the presence of statements like \( m \neq m \), which can be easily detected (see, for instance, literature about the alldifferent constraint, [Reg94]).

Practically, each disjunct is checked, by progressively instantiating its symbolic variables, as follows:

- **each positive literal**, i.e. \( x = m \) or \( \kappa \triangleright m \), may impose a finite number of necessary conditions, corresponding to one or more concretising substitutions, via a sort of symbolic unification. For example \( xa_1(\{\{noa_1\}_k,\{nob_1\}_k\}) = \{ya_1(\kappa)\}_k \) can be concretised by mapping the symbolic variable \( ya_1(\kappa) \) either in \( nob_1 \), or in \( noa_1 \), provided that they can be derived from \( \kappa \). A lazy strategy, i.e. instantiating only as much as necessary, may be adopted to prevent combinatorial state explosion. Not always does a “unifying” substitution exist, and this leads to the failure of the verification process for the positive literal under consideration, and hence for the conjunction in which it occurs. Each existing substitution is applied to all the other literals. When no other substitutions can be generated, the process for positive literals ends. (This procedure mimics the Martelli-Montanari non-deterministic unification algorithm, [MM82]).

- **the negative literals**, i.e. the inequalities over infinite domains \( x \neq m \) and \( \kappa \ntriangleright m \), can be dealt with by standard constraint solving techniques, as explained above. For example the literal \( xa_1(\{\{k\}\}) \neq yb_3(\{\{k\}\}) \) has always a solution (i.e. choosing two different messages among the infinite many available), while the literal \( xa_1(\{\{k\}\}) \neq xa_1(\{\{k\}\}) \) has not.

A formula is satisfiable, if for any of its disjuncts the corresponding constraint system has a concretising substitution \( \psi \) as a solution. In this case \( \kappa \psi \models_{\chi;\psi} \phi \).

Trivially, the algorithm presented above always terminates, being based on known terminating transformations. Moreover, its correctness can be guaranteed by construction, i.e. truth preserving transformations, by exhaustive non-deterministic exploration and by the correctness and completeness of constraint solving techniques.

From soundness of the symbolic semantics, it follows that one can symbolically simulate the protocol, without errors (or loss of information), and then correctly
validate the property expressed by a symbolic formula, obtaining a symbolic model that implies one, or more, concrete model. In a word, “symbolically reasoning” is safe.

Example 5.11 The presence of inequalities in the formula gives rise to an infinite number of possible concrete models. For example, the formula $xa_i({no_1, no_2}) \neq yb_j({no_2, no_3})$, which is satisfiable, can produce different concrete models by means of the substitution

$$\{xa_i({no_1, no_2}) \mapsto a, yb_j({no_2, no_3}) \mapsto no_3\},$$

with $a \in \{no_1, no_2\}$, but also by means of

$$\{xa_i({no_1, no_2}) \mapsto (no_2, \ldots, no_2), yb_j({no_2, no_3}) \mapsto no_3\}.$$

Typically, in this case, in order to attack the protocol, the message generated for $xa_i$ is not important as itself, but as being different from $yb_j$.

5.7.6 Symbolic verification: An example

This section illustrate the second part of the verification methodology, concluding what started in Section 5.7.4. There, symbolic analysis has been applied to the the construction of a symbolic trace for the protocol:

$$S \rightarrow A: \{k\}_{A^+} \quad S \triangleq (xs, ys)[out({k}_{xs^+}).out({k}_{ys^+})]$$
$$S \rightarrow B: \{k\}_{B^+} \quad A \triangleq ([in({?ua})_{A^-}).in({?wa})_{ua}]]$$
$$B \rightarrow A: \{no\}_k \quad B \triangleq ([in({?vb})_{B^-}).out({no}_{vb}])$$

Here, we show how to formalise a security property of the protocol by means of a formula of $\mathcal{PL}$, and how to verify the formula with respect to the terminating symbolic trace previously constructed.

The intended meaning of the protocol is that $S$ provides a safe session key $k$ to the principals $A$ and $B$. We can consider the key safe if the message $m$ it encrypts can not be known by an intruder. Moreover, we can also require that if $S$ has sent $k$ to $B$ (i.e. either the open variable $xs$ or $ys$ is linked to $B$), then the message that $A$ receives in $wa$ is the one that $B$ has sent, i.e. $no$. Considering that we require these two properties holding for any instance participating in a multi-session of the protocol, we can formalise the correctness of the protocol as:

$$\theta \equiv \forall i : B \exists j : S \exists h : A \not\equiv no_i \land ((ys_j = B_i \lor xs_j = B_i) \rightarrow wa_h = no_i),$$

from which the protocol is not safe if an intruder can drive it to a terminating state where it holds.

$$\phi \equiv \neg \theta \equiv \exists i : B \forall j : S \forall h : A. \not\equiv no_i \lor ((ys_j = B_i \lor xs_j = B_i) \land wa_h \neq no_i).$$
In Section 5.7.4, we have constructed a trace which, starting with the knowledge 
\( \kappa_0 = \{ I_0, I_0^-, kk, mm_0 \} \), drove a session with an instance of each protocol principal 
to the final state \( \langle \{ \}, \chi, \kappa \rangle \), where 
\[
\chi = \{ x_1 \mapsto A_2, y_1 \mapsto B_3, u_2 \mapsto kk, v_3 \mapsto kk, w_2 \mapsto w_2(\kappa) \}
\]
\[
\kappa = \{ I_0, I_0^-, mm_0, kk, S_1, A_2, B_3, \{ k \} _{A_2^+}, \{ k \} _{B_3^+}, \{ no_3 \} _{kk}, no_3 \}
\]
and the knowledge corresponding to the symbolic variable \( w_2 \) corresponds to the 
final intruder knowledge.

As explained in Section 5.7.5, satisfiability of \( \phi \) is checked in a normalised form: 
a disjunction of conjunction of literals, from which quantifiers have been eliminated. 
In our case, 3 is the only admissible index for \( i : B \), 1 for \( j : S \), and 2 for \( h : A \). The 
formula becomes:
\[
\phi' \equiv \kappa \triangleright no_3 \lor (y_1 = B_3 \land w_2(\kappa) \neq no_3) \lor (x_1 = B_3 \land w_2(\kappa) \neq no_3). 
\]
Two more steps are necessary in verifying \( \phi' \):

1. Actualising \( \phi' \) to a final state of a trace by applying the substitution \( \chi \), which 
resumes the history of the trace. In our case:
\[
\phi' \chi \equiv \kappa \triangleright no_3 \lor (B_3 = B_3 \land w_2(\kappa) \neq no_3) \lor (A_2 = B_3 \land w_2(\kappa) \neq no_3). 
\]

2. Determining, by constraint solving, a concretization which satisfies the 
formula. In our case the formula is trivially true because of \( \kappa \triangleright no_3 \). Let us 
consider anyway the term \( (B_3 = B_3 \land w_2(\kappa) \neq no_3) \), its satisfiability depends 
on the possibility for \( \kappa \) to generate a message different from \( no_3 \), that surely 
holds. The third disjunct does not hold.

The intruder, by substituting the server \( S \) in providing a session key to the 
principals \( A \) and \( B \), has been able both to discover their secret \( (\kappa \triangleright no_3) \) and to 
interfere in their communication by sending a different value from the expected one 
\( (w_2(\kappa_8) \neq no_3) \).

## 5.8 Related work

Protocol verification literature consists of a relevant number of different proposals, 
applying different techniques. Some of them involve state space exploration via 
model checking [CJM98, Low95, MSS98]. Other approaches rely on inductive theorem 
proving [Pau98], and on logic programming [Mea96]. All these approaches 
made simplifying hypothesis on the behaviour of the intruder to bound the state 
space.

Several logical formalisms have been proposed to specify properties of security 
protocols. Here, we mention the BAN logic [BAN90] and the correspondence 
relations of Dolev and Yao [DY83]. In the BAN logic a protocol \( P \) is idealized so as to
formally state the intended meaning of messages exchanged by principals. However, this is a non trivial task and requires a strong 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 protocol is automatically checked. The Dolev and Yao correspondence relations have been adopted in [CJM98, Hui99, Bor01]. A correspondence, written as $\alpha \leftarrow \beta$, asserts that, for all traces, each occurrence of an action $\alpha$ must be preceded by an occurrence of an action $\beta$. 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, so that it is not easy to express a property regarding a generic number of sessions.

Process algebras have been extensively adopted to specify and verify properties of cryptographic protocols. Well known approaches which exploit variations of CCS or CSP are described in [FG95, FG97, Low97, Low99, Sch96, Sch98, RS99]. In the last few years variations of the $\pi$-calculus (e.g. the spi-calculus [AG98, AG99]) have been employed to specify and verify properties of security protocols. Unlike CCS/CSP-like calculi, the spi-calculus provides basic constructs for the generation of new names (nonces) and keys.

The cIP calculus has many commonalities with the $\pi$-calculus and the spi-calculus. Indeed, the notion of “magic” instance of a protocol introduced in [AG99] has been one of the inspiring idea of our work. Using that approach, based on the spi-calculus, integrity and secrecy properties of protocols are verified via semantic equivalence ($\cong$) between the protocol and one of its instances, which magically satisfies the property. The $PL$-logic is based on similar principles, but verification is done by means of model checking and the set of properties it can express is extended. Another difference with spi-calculus, is that cIP does not present cryptographic actions, being encryption and decryption embedded in communications as an extension to the concept of duality of input/output actions. We believe that being closer with the original calculus permits the well settled theory of $\pi$-calculus to be more easily “imported” in the new language.

As far as model checking applied to protocol verification is concerned, [CJM98] has been inspiring for our work. With respect to that proposal, where message derivation was proved decidable in a natural deduction style, we have extended the framework to the non-trivial case of asymmetric keys. Moreover, by means of symbolic analysis techniques, it has been possible to overcome their limitation about the maximum complexity of the messages exchanged in a protocol.

The cIP calculus shares also similarities with the calculus introduced in [Bor01]. There, all the processes interact by means of a unique public channel and every message sent over it increases the “environment” knowledge. That calculus relies on a symbolic semantic which exploits unification to restrict the state space (traces) of process behaviour. Protocol analysis is obtained through a search for a (symbolic) trace that respects some properties. The uniqueness of the public channel is a
key feature of both the calculi. As well, the knowledge of the intruder (in a give instant) is represented by all the messages that have been sent along the public channel. Our verification procedure also takes symbolic traces, but differently the analysis is driven by constraints. Moreover, our approach permits us to directly deal with multi-sessions, while in [Bor01], multiple sessions must be “hand-written” by a human verifier.

Finally, the logic presented in [Hui99] has many similarities with the $\mathcal{PL}$-logic. However, unlike our approach, that logic lacks of quantifiers over protocol sessions, and, it cannot deal directly with multiple sessions. Hence, it suffers of the same “hand-writing” problem of [Bor01].

The direct handling of multiple sessions is a distinguished feature of our approach. The relevance of multi-session attacks has been enlighten in [Mil99]. Some results [Low98] had proven that, in particular cases, it is possible to impose sufficient conditions that allow one to limit the analysis to “small system”, i.e. sessions where few protocol sessions are present. However, those results are not general and only hold for secrecy properties.

\section{Concluding remarks}

In this chapter we developed a methodology to verify correctness of multi-session security protocols which run in an open environment. The distinctive features of our approach provide natural mechanisms for modeling the dynamic composition of principals and verifying its properties. The name-based framework we presented, applied to cryptographic key sharing, provides a contribution to the general issues of the secure composition of components into open systems.

The IP-calculus has been easily extended so as to be applied to the formal verification of security protocols in a quite rich setting, supporting symmetric and asymmetric key protocols for which new results have been proved (namely, the decidability of message derivation), featuring multi-sessions verification, allowing a significant set of properties to be expressed. The $\mathcal{PL}$-logic represents an expressive means for declaratively reasoning about data exchanged and roles played by the principals in the protocols, while symbolic analysis permits the verification of the logical formulas to be automatically and effectively performed.

There are several directions in which the work reported here is worth being extended.

Currently we are finalising the grounding of the theory as a detailed formal specification for the ongoing development of the ASPASYA tool (after Automated tool for Security Protocol Analysis based on a Symbolic Approach), which provides a computational implementation of the symbolic protocol analysis presented in this chapter. The tool, implemented in OcctML, gave in a preliminary testing phase encouraging results, both in terms of expressiveness and performances with respect to state of the art existing tools, and it seems amenable to further enhancements.
5.9. CONCLUDING REMARKS

Other than the facts that the cIP-calculus is well-suited for dealing with security concerns and that every formalisation of open systems needs to deal with the security aspects of communication, the application of IP-calculus to security finds another motivation in studying composition of components subject to the satisfaction of properties of the resulting system.

The property $P(S, P, \gamma, S')$ of rule $\text{join}$ in Figure 3.5 (Chapter 3) was meant to express a generic condition that the interaction pattern $P$, the session $S$, the chosen mapping $\gamma$ and the session $S' = \text{join}(P, \gamma, S)$ should satisfy in order to let $P$ join the session $S$. A first definition of the property $P$ has been given in the rule $(\text{in})$ of Definition 3.9, where the requirement was that $P$ would not have spoiled the acceptability of $S$. As introduced by the motivations presented in the preface of this chapter, it appears natural to try to increase the expressiveness of the join mechanism with constraints enforcing the desired properties of a system. Starting from security issues, according with that introduced in [BBFT02], the idea is to associate a policy to a session $S$ (possibly as a consequence of an off-line analysis of its security requirements), and then to map this policy over constraints $\psi$ on the sharing of references. Hence, $\text{join}(S, \gamma, P)$ yields the new session $S'$ only if the mapping $\gamma$ respects the policy $\psi$, informally $S' \models_{\gamma} \psi$. Note that the relation $\models$ is not meant to require a model checking of traces, but rather the much simpler checking of satisfiability of a constraint over shared references, and hence it can be more efficiently verified, in particular for dynamic system reconfiguration.

In [BBFT02], we have shown how the correctness a well-known protocol (the Needham-Schroeder protocol) can be implied by a condition over the sharing of keys, that, informally speaking, sounds like “every principal, that has been engaged to play a given part in the protocol, can not acquire the keys used by the principals playing other parts” (even if it would not use them to break the protocol). By associating this constraint to the join operation, any principal attempting to concurrently play more roles in the protocol would be rejected. Hence also an intruder, which intermixes two sessions, would not be able to join the session. Note that, while the stronger property may prevent “honest” principal to join the session, it can, on the other hand, be more efficiently checked.

A further experiment in this direction has been done in the implementation of ASPASYA, where properties about the identities of the principals, their number, and the initial sharing of keys have been used to remarkably reduce the search space in the generation of significant traces.

The expressiveness of the logic deserves further investigation, especially concerning a more precise characterisation of the security properties that it is able to express, other than the well-known integrity, secrecy and authentication.

It would also be interesting to verify whether this framework can help in the understanding and construction of an hierarchy of security properties, a currently open research problem. The generalisation to property for composition and coordination of components in open systems, straightforwardly follows (as partially shown in [BBFT02]) as a future line of research.
5.10 Proofs of Chapter 5

Proof of Theorem 5.2 \([e(\_)] \) is well defined]

Proof.

The function always terminates. It is enough to consider the number of parenthesis and brackets in \( \kappa \). Either \( e(\_ \,) \) is recursively called on a set having a strictly smaller number of parenthesis and brackets, or it returns the set itself. The number can not become negative and hence the recursion terminates.

The function is deterministic. The function has a non-deterministic definition, since the conditions \( m = (p, q) \in \kappa \) and \( m = \{n\}_\lambda \in \kappa \wedge \lambda \in \kappa \) are not mutually exclusive. It must be shown that, despite the possible choices for the recursive calls, \( e(\_ \,) \) returns a unique value. It is necessary to prove that, given a knowledge \( \kappa \)

\[ \exists \ \kappa_1, \kappa_2, \ k_1 = e(\kappa) \wedge k_2 = e(\kappa) \Rightarrow k_1 = k_2 \]

Let us assume, by absurdum and without loss of generality, that \( \exists \ m. \ m \in k_1 \wedge m \not\in k_2 \). The proof proceeds by induction on the definition of \( e(\_ \,) \). Assuming that the recursive calls of \( e(\_ \,) \) over smaller knowledges are well defined, it is possible to show that \( e(\_ \,) \) is well defined. In the rest of the proof the observation that if \( m \in \kappa \), then \( m \in e(\kappa) \) will be used (in fact no rule of the definition of \( e(\_ \,) \) eliminates an item of \( \kappa \)). The proof is given by cases on the construction of \( k_1 = e(\kappa) \).

1. \[ k_1 = e(\kappa) = \kappa \] \hspace{1cm} (5.1)

The other rules can not applied and hence the only outcome of \( e(\kappa) \) is \( \kappa \) itself.

2. \[ k_1 = e(\kappa) = e(\kappa \setminus \ o \cup p \cup q) \cup o \] \hspace{1cm} (5.2)

Analogously, \( k_2 \) can be constructed from \( \kappa \) in two different ways.

(a) \( k_2 = e(\kappa) = e(\kappa \setminus \ o' \cup p' \cup q') \cup o' \)

Obviously \( o \neq o' \), otherwise, applying the inductive hypothesis, \( k_1 = k_2 \). Since \( m \in k_1 \), from (5.2), either \( m = o \) or \( m \in e(\kappa \setminus o \cup p \cup q) \).

In the first case, by the side conditions of rule (5.2), \( m \in k, \) and hence \( m \in k \setminus o' \cup p' \cup q' \) and finally \( m \in e(\kappa \setminus o' \cup p' \cup q') \subset k_2 \). Absurdum.

On the other hand, assuming \( m \neq o \in e(\kappa \setminus o \cup p \cup q) \), from rule (5.2), it follows that \( m \in e(\kappa) \setminus o \). Recalling that \( m \neq o' \), it also follows that \( m \in e(\kappa) \setminus o' \subset k_2 \). Absurdum.
5.10. PROOFS OF CHAPTER 5

185

5.10.1 Proof of Theorem 5.3 \([\kappa \triangleright m \text{ is decidable}]\)

Proof.

\[ \kappa \triangleright m \Rightarrow e(\kappa) \triangleright_1 m \]

(by induction on \(i\), the length of the proof for \(\kappa \triangleright m\)).
\[ i = 1. \]

The only proof of length one is
\[
\frac{m \in \kappa}{\kappa \vdash m \in},
\]
and then, by Lemma 5.1, \( m \in e(\kappa) \), and hence \( e(\kappa) \triangleright_i m \).

\[ i \Rightarrow i + 1. \]

The message \( m \) is derived by a proof which can terminate by applying one of the remaining five rules.

If \( m = (p, q) \) is obtained by applying the rule
\[
\frac{\kappa \triangleright p \quad \kappa \triangleright q}{\kappa \triangleright (p, q) \in},
\]
then, by inductive hypothesis \( e(\kappa) \triangleright_i p \) and \( e(\kappa) \triangleright_i q \), and hence
\[
\frac{e(\kappa) \triangleright_i p \quad e(\kappa) \triangleright_i q}{e(\kappa) \triangleright_i (p, q) \in}.\]

If \( m = \{n\}_\lambda \) is obtained by applying the rule
\[
\frac{\kappa \triangleright n \quad \kappa \triangleright \lambda}{\kappa \triangleright \{n\}_\lambda \in},
\]
then, by inductive hypothesis \( e(\kappa) \triangleright_i n \) and \( e(\kappa) \triangleright_i \lambda \), and hence
\[
\frac{e(\kappa) \triangleright_i n \quad e(\kappa) \triangleright_i \lambda}{e(\kappa) \triangleright_i \{n\}_\lambda \in}.\]

If \( m = p \) is obtained by applying the rule
\[
\frac{\kappa \triangleright (p, q) \quad \kappa \triangleright p \in}{e(\kappa) \triangleright_i (p, q) \in},
\]
then, by inductive hypothesis \( e(\kappa) \triangleright_i (p, q) \). The proof for this last judgment can be concluded in two ways. A case occurs when applying the rule
\[
\frac{e(\kappa) \triangleright_i p \quad e(\kappa) \triangleright_i q}{e(\kappa) \triangleright_i (p, q) \in},
\]
which implies that \( e(\kappa) \triangleright_i q \). The other case occurs when the rule
\[
\frac{(p, q) \in e(\kappa)}{e(\kappa) \triangleright_i (p, q) \in}
\]
is applied. If \( (p, q) \in e(\kappa) \), from the definition of \( e(\kappa) \), it follows that also \( p \in e(\kappa) \) (if not the recursive construction of \( e(\kappa) \) could not have terminated). It follows that \( e(\kappa) \triangleright_i p \).
The case for $(\cdot)_{e2}$ is symmetric to the previous one.

The last case is analogous to the previous two. If $m = n$ is obtained by applying the rule

$$
\frac{\kappa \vdash \{n\}_\lambda \quad \kappa \vdash \lambda^-}{\kappa \vdash n \quad \{\}_e},
$$

then, by inductive hypothesis $e(\kappa) \triangleright_i \{n\}_\lambda$ and $e(\kappa) \triangleright_i \lambda$. The proof for $e(\kappa) \triangleright_i \{n\}_\lambda$ can be concluded in two ways. A case occurs when applying the rule

$$
\frac{e(\kappa) \triangleright_i n \quad e(\kappa) \triangleright_i \lambda}{e(\kappa) \triangleright_i \{n\}_\lambda \quad \{\}_i},
$$

which implies that $e(\kappa) \triangleright_i n$. The other case occurs when the rule

$$
\frac{\{n\}_\lambda \in e(\kappa)}{e(\kappa) \triangleright_i \{n\}_\lambda \quad \in}
$$

is applied. If $\{n\}_\lambda \in e(\kappa)$, from the definition of $e(\_)$, it follows that also $n \in e(\kappa)$ (if not the recursive construction of $e(\_)$ could not have terminated). It follows that $e(\kappa) \triangleright_i n$.

$e(\kappa) \triangleright_i m \Rightarrow \kappa \triangleright m$

(by induction on $i$, the length of the proof for $e(\kappa) \triangleright_i m$).

$i = 1$.

The only proof of length one is

$$
\frac{m \in e(\kappa)}{e(\kappa) \triangleright_i m \quad \in},
$$

and then, by Lemma 5.2, $\kappa \triangleright m$.

$i \Rightarrow i + 1$.

Two rules can be applied to derive $e(\kappa) \triangleright_i m$ in more than one step, namely $(\cdot)_i$ and $\{\}_i$. In the first case $m = (p, q)$, and

$$
\frac{\kappa \triangleright_i p \quad \kappa \triangleright_i q}{\kappa \triangleright_i (p, q) \quad (\cdot)_i},
$$

then, by inductive hypothesis $\kappa \triangleright p$ and $\kappa \triangleright q$, and hence

$$
\frac{\kappa \triangleright p \quad \kappa \triangleright q}{\kappa \triangleright (p, q) \quad (\cdot)_i}.$$
Analogously, if \( m = \{ n \}_\lambda \) and \( \{ \}_i \) is used, then
\[
\frac{e(\kappa) \triangleright_i n \quad e(\kappa) \triangleright_i \lambda}{e(\kappa) \triangleright_i \{ n \}_\lambda}
\]
\( \{ i \} \).

By inductive hypothesis \( \kappa \triangleright n \) and \( \kappa \triangleright \lambda \), and hence
\[
\frac{\kappa \triangleright n \quad \kappa \triangleright \lambda}{\kappa \triangleright \{ n \}_\lambda}
\]
\( \{ i \} \).

\[\square\]

**Proof of Lemma 5.2** \([ m \in e(\kappa) \Rightarrow \kappa \triangleright m ]\)

**Proof.**

(by structural induction on \( e \)).

The set \( e(\kappa) \) can have been obtained by applying one of the three rules of the definition of the function \( e(\_ \) .

\( e(\kappa) = \kappa \)

In this (base) case, \( m \in \kappa \) and hence, by applying rule \( \in \), \( \kappa \triangleright m \).

\( e(\kappa) = e(\kappa \setminus o \cup p \cup q) \cup o \)

The side conditions for this case imply that \( o = (p, q) \in \kappa \). Either \( m = o \in \kappa \), but then \( \kappa \triangleright m \) (by rule \( \in \)), or \( m \in e(\kappa \setminus o \cup p \cup q) \). By the inductive hypothesis, it follows that \( \kappa \setminus o \cup p \cup q \triangleright m \), and equivalently \( \kappa \cup p \cup q \triangleright m \). Note that, since \( o = (p, q) \in \kappa \), by applying \( (\_)_e \) and \( (\_)_e \), it also holds \( \kappa \triangleright p \) and \( \kappa \triangleright q \). Trivially (reasoning on the proofs):

\[
\begin{align*}
\kappa \triangleright p \\
\kappa \triangleright q \\
\kappa \cup p \cup q \triangleright m
\end{align*}
\]

\[\Rightarrow \kappa \triangleright m \]

\( e(\kappa) = e(\kappa \setminus o \cup n) \cup o \)

Analogously, \( o = \{ n \}_\lambda \in \kappa \) and \( \kappa \triangleright \lambda \) (and, obviously, \( \kappa \triangleright \lambda \)). If \( m = o \in \kappa \), then \( \kappa \triangleright m \). Otherwise, \( m \in e(\kappa \setminus o \cup n) \), and then, by inductive hypothesis, \( \kappa \cup n \triangleright m \). Since \( o \in \kappa \), \( \kappa \triangleright n \) by \( \{ \}_e \). Trivially:

\[
\begin{align*}
\kappa \triangleright n \\
\kappa \cup n \triangleright m
\end{align*}
\]

\[\Rightarrow \kappa \triangleright m \]

\[\square\]
5.10.2 Proof of Theorem 5.4

\[ \forall \psi \Sigma_0 \sim^n \Sigma_n \Rightarrow \Sigma_0 \hookrightarrow^n \Sigma_n \psi \]

Proof.

(by induction on \( n \), the length of the symbolic trace \( \Sigma_0 \sim^n \Sigma_n \))

\( n = 0 \).

Trivial.

\( n - 1 \Rightarrow n \).

Let us suppose that \( \Sigma_0 \sim^{n-1} \Sigma_{n-1} \). The \( n \)th step of the trace can be done according to one of the three rules of symbolic semantics (Figure 5.4). We show the most significant case of rule \((I - \text{out})\) (the others work analogously), where the intruder generates a symbolic message for a principal. We assume that this is done according to the procedure described in Section 5.7.3, in which case it is easy to show, by structural induction on \( m', \) that \( \mu(m', \kappa) = (m, \sigma) \Rightarrow \kappa \triangleright m \land m\sigma \sim_s m'\sigma \). Then (with \( m \) and \( \sigma \) generated by \( \mu \))

\[
\Sigma_{n-1} = \langle \{(X_i)[E_i]\} \cup C, \chi, \kappa \rangle \triangleright^{(m\sigma)} \langle \{(X_i)[E'_i]\} \cup C, \chi; \sigma, \kappa \sigma \rangle = \Sigma_n
\]

\((I - \text{out})\)

Let \( \psi \) be a concretising substitution for \( \Sigma_n \). In order to prove the theorem, we choose one of the possible concretising substitutions \( \tilde{\psi} \) such that

\[
\Sigma_0 \hookrightarrow^{n-1} \Sigma_{n-1} \tilde{\psi} = \langle \{(X_i)[E_i]\} \tilde{\psi} \cup C, \chi \tilde{\psi}, \kappa \tilde{\psi} \rangle,
\]

whose existence is granted by the inductive hypothesis, and a suitable substitution \( \tilde{\sigma} \) that allows the concrete n-th step to occur. In particular, we choose \( \tilde{\psi} = \sigma \psi \) over all (and only) the symbolic variables in \( \text{dom}(\sigma \psi) \), i.e. \( \tilde{\psi} \) concretises coherently with what done by \( \psi \) at the n-th step of the symbolic trace, and \( \tilde{\sigma} = \sigma \psi \) over all (and only) the non-symblic variables in \( \text{dom}(\sigma \psi) \), i.e. \( \tilde{\sigma} \) lets data matching at the n-th concrete step coherently with the n-th symbolic step and the subsequent concretisation due to \( \psi \).

The following facts hold:

1. \( \tilde{\sigma} \tilde{\psi} = \sigma \psi \); since, by construction, \( \tilde{\sigma} \tilde{\psi}(x) = \tilde{\sigma}(x) = \sigma \psi(x) \), for \( x \in V \), and \( \tilde{\sigma} \tilde{\psi}(x) = \tilde{\psi}(x) = \sigma \psi(x) \), for \( x \in V(\kappa) \), being \( x \in \text{dom}(\tilde{\sigma} \tilde{\psi}) = \text{dom}(\sigma \psi) \).

2. \( \tilde{\sigma} \tilde{\psi} = \tilde{\psi} \tilde{\sigma} \); since no variable occurs in their ranges and their domains are disjointed.
3. $E_i \psi \xrightarrow{\text{in}(m' \bar{\psi})} E'_i \bar{\psi}$, by construction.

4. $\kappa > m \Rightarrow \kappa \bar{\psi} > m \bar{\psi}$, by definition of $>$ and $\bar{\psi}$.

5. $m \sigma \sim m' \sigma \Rightarrow m \sigma \psi \sim m' \sigma \psi$, since matching of symbolic messages is preserved by concretising them (by structural induction on $m$).

6. $m \sigma \psi \sim m' \sigma \psi \Rightarrow m \sigma \bar{\psi} \sim m' \sigma \bar{\psi}$, by 1.

7. $m \sigma \bar{\psi} \sim m' \sigma \bar{\psi} \Rightarrow m \sigma \bar{\psi} \bar{\sigma} \sim m' \sigma \bar{\psi} \bar{\sigma}$, by 2.

The highlighted facts justify the following transition:

$$
\Sigma_{n-1} = \langle\{(X_i)[E_i] \bar{\psi}\} \cup \mathcal{C}\psi, \chi \psi, \kappa \psi\rangle \mapsto \langle\{(X_i)[E'_i \sigma \psi]\} \cup \mathcal{C}\sigma \psi, \chi; \sigma \psi, \kappa \sigma \psi\rangle = \Sigma'_n
$$

Finally, from 1 and 2 it holds that

$$
\Sigma'_n = \langle\{(X_i)[E'_i \sigma \psi]\} \cup \mathcal{C}\sigma \psi, \chi; \sigma \psi, \kappa \sigma \psi\rangle = \Sigma_n \bar{\psi},
$$

proving the inductive step, and then the theorem.
Chapter 6

Conclusions

Nowadays, and increasingly in the immediate future, software applications are and will continue to be conceived, designed and run in a distributed world of autonomous components, where a significant amount of the work is consumed by components to properly interact with each other and coordinate their tasks. Life in this world is complicated by its being highly dynamic, and hence only partially accessible at a given instant. Moreover, also the possible presence of malicious components which may try to exploit the others and take advantage form interacting with them must be taken into account.

The ongoing development of this technology requires the development of new instruments to cope with the intrinsic complexity of such a scenario.

The work contained in this dissertation has aimed to develop a formal model of the interaction of components, to understand better, design and verify the applications that are living in the above mentioned world, a world which is going to be more and more intertwined with our everyday world.

A better understanding of the interaction and coordination of components has both a practical and theoretical interest, since, as discussed, current technology lacks effective means to systematically afford component interaction, and as such, several interaction aspects are research topics currently under investigation.

We propose a methodology for modeling local properties of open systems, in order to manage the fact that systems may be only partially specified, and also to improve the effectiveness of the method. Here \textit{local} means temporally limited: the system is observed to a limited extent of its potentially infinite life, namely, the part that is fully specified. Component behaviour is projected over finite fragments that are composed together in an open environment. The model uniformly applies to closed and open systems, as well as to static and dynamic kinds of analysis. The name-based features of the calculus adopted permit us to precisely characterise component composition and their subsequent interaction. On this basis, it has been shown how to verify and enforce local correctness properties, throughout the life of an open system. Correctness of a system is obtained as an invariant guaranteed by preventing the access to the system of any potentially harmful component.
The model has then been used to address the problem of component adaptation, and a methodology for constructing adaptors, i.e. components that facilitate the interoperability of other mismatching components, has been devised exploiting the existing features of the model. Correctness, hence, is no longer obtained by forbidding interaction among mismatching components, but, rather by facilitating their interaction towards successful termination.

Security issues have also been addressed within our methodology, since they are a key topic for open systems. In particular we have studied the formal verification of security protocols. Our model is well suited to deal with security protocols that consist of a finite interaction among a set of components, the principals of the protocol, while the openness of the system models well the possibility of the existence of an intruder. Moreover, the properties that security protocols are meant to enforce can be formalised, by means of an expressive logic language we defined, as invariant properties of the system holding after the termination of (a number of) protocol execution.

This dissertation is concluded by considering the results obtained by investigating the above mentioned topics. The work done opens, in several cases, interesting perspectives for further analysing the issues already addressed and for more general future research. Results and future work are summarised together in the following.

**Modeling the behavioural aspects of open systems**

We devised a model for open systems, based on an original, temporally finite representation of component behaviours as interaction patterns. Thanks to the finiteness of the approach we can check a local correctness property of an open system despite its being only partially specified. Finiteness reduces the high computational costs typical of the process algebra based approaches, and also allows for further controlling it by choosing a proper dimension of the behavioural patterns. On the other hand, part of the expressiveness of a $\pi$-calculus based language is retained, most importantly the possibility to dynamically reconfigure the communication network.

We have proposed an original correctness property for open systems, namely acceptability, that within the formal model can be easily verified and guarantees the “so-far” correctness of the system, or in other words that up to now the system does not present critical errors. Acceptability, then, establishes an effective way of ruling the dynamical access to a running system, excluding those components which may spoil its correctness.

The definition of an algorithm for checking acceptability provides the computational basis on which the theoretic model can be developed in an actual software architecture. The structure of the algorithm, and in particular the
visiting strategy of the state space, gives interesting insights into how to incrementally build a completion, in case of acceptability, and, more in general, a component which is “compatible” with its environment.

The model would benefit from being tested in the large on some component architecture. A first step consists in extending traditional IDLs of a middleware (e.g. CORBA) or component (e.g. JavaBeans) architecture with behavioural concerns, according to the proposed model.

There can be many completions of a session. It is interesting to study a hierarchy of them in order to define a representative completion. Such completion could be used as a measure of the complexity of the session. The existing studies surveyed, about behavioural inheritance and compatibility, constitute a valuable starting point.

Sessions are flat, in the sense that a session can not contain other sessions. An immediate line of research is to investigate if it is worth adding a structure to sessions, à la ambient calculi, and to study which systems can be modeled with this extension. In particular it seems reasonable that components in a session are distinguished according to their importance for the life of the overall system, and that for different sets of components different properties are required.

**Adapting the mismatching behaviour of components**

We have developed adaptation within IP-calculus, against most of the known proposals which prefer FSM for efficiency reasons. While finiteness of our approach guarantees a comparable computational complexity, the use of a more expressive formalism allows for a richer set of cases to be dealt with.

More significantly, we have defined an abstract language for describing the required adaptation. Adaptation hence consists of a more declarative mapping specification phase, in which the task of the designer is facilitated by a high-level language, and a more operational adaptor construction phase, which is delegated to an algorithmic procedure. Another distinguishing feature of our approach regards the extra properties that an adaptor can be required to satisfy: while in other approaches such properties are used to verify a constructed adaptor, in our framework they can be used in the specification of the adaptor. Minor changes are necessary to the algorithm to produce an adaptor fulfilling the extra desired properties.
At the moment all the adaptors are characterised as components which fulfill the mapping and other possible properties. It is interesting, as in the case of completions, to understand how all the correct adaptors relate to each other. In particular, it would be valuable to characterise adaptors in terms of the amount of facilitation they provide.

Implementations can be tested within the testing of the general framework previously mentioned.

**Verifying security protocols**

The language, in the spirit of the analogy “protocols as sessions”, permits us to easily deal with multi-session protocol runs, where a session is repeatedly joined by more instances of the protocol principals. Accordingly, the logic developed to express security properties easily quantifies over protocol instances. The framework, uniformly, deals with both asymmetric and symmetric keys. The facility of expressing properties regarding an unbound number of principal instances results in being an expressive means of verifying protocols against multi-session attacks, a relevant case of protocol analysis, and a feature distinguishing our proposal from the vast majority of the others.

Symbolic analysis is recognised to be the proper means to deal with infinite branching problems. Differently from other cited proposals adopting it for protocol verification, our symbolic analysis seems to be closer to concrete semantic in terms of a stricter correspondence of symbolic and (terminating) concrete traces. This point still requires further investigation.

The logic is based on the implicit assumption that the security properties can be expressed as relations among data transmitted, principals of the protocol and knowledge of the environment. An immediate topic for further investigation is the expressiveness of the logic. This task is complicated by the lack of a precise hierarchy of the security properties, which is also an open problem worth being investigated.

As for the other aspects of the model, testing the methodology by the verification of a large number of currently used protocols is a direction to pursue. At this aim, the ASPASYA tool is currently under development and some preliminary tests have been performed.

Finally, we want to conclude with two considerations, suggesting more general directions for further research encompassing both behavioural and security aspects of open systems.
A logic predicating over data, components, and their relations may likely be used to express more general properties about component compositions in open systems.

As indicated, and partially developed in some recent papers, security properties, and more general properties, can be enforced by constraints over the way components are connected together. More generally, it appears interesting, especially for the aims of dynamic verification, to study how to deduce behavioural properties of a system from the interconnection topology of its components, analogously to what has been proposed by some of the surveyed work.
Bibliography


[ALSN00] F. Achermann, M. Lumpe, J. Schneider, and O. Nierstrasz. Piccola - a small composition language. In Howard Bowman and John Derrick,


[BDNN00] C. Bodei, P. Degano, F. Nielson, and H.R. Nielson. Static analysis for
the $\pi$-calculus with applications to security. Information and Compu-

[Ber98] K.L. Bernstein. A congruence theorem for structured operational se-
manitics of higher-order languages. In Proceedings 13th LICS Sympo-

coordination primitives. Information and Computation, 156(1/2):90–


[BJ99] A. Brogi and J.M. Jacquet. On the expressiveness of coordination mod-
els. In P. Ciancarini and L. Wolf, editors, Coordination languages and
models: Third International Conference, pages 134–149, Amsterdam,

model: ten years after. In J.-M. Andreoli, H. Gallaire, and D. Le
Mêtayer, editors, Coordination programming: mechanisms, models and

[Bor01] M. Boreale. Symbolic trace analysis of cryptographic protocols. In
28th Colloquium on Automata, Languages and Programming (ICALP),

Workshop on Component-Oriented Programming (WCOP’97), pages

the secrecy goals. Lecture Notes in Computer Science, 1485:361–375,
1998.

[BPSM97] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Lan-


[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, 1994.


Index

Acceptability checker algorithm, 72
deadlocks, 74
evolutions_after, 76
nodes, 74
success, 74
triggers, 76
code, 77
complexity, 82
correctness, 84
ADAPT algorithm, 107
M_triggers, 106
code, 108
ℓ(m), 168
κ \models^s \varphi, 176
κ \models^x \varphi, 153
\mathcal{PL}
protocol logic, 150, 152
model, 153
symbolic model, 176
μ(n,κ), 168
ν(m, n), 170
\triangleright, 164
IP-calculus, 40
semantics, 44
in-rule, 64
approximation relation ≪S, 86
asymmetric cryptographic algorithms, 124
attack, 155
behavioural expression, 41
certificates, 124
cIP-calculus, 137
bound and free vars, 140
concretising substitution, 165
data, 138
indexed data, 141
indexed message, 141
indexed variable, 141
matching, 145
open variables, 139
symbolic data, 164
symbolic knowledge, 164
symbolic message, 164
symbolic substitution, 165
symbolic variable, 164
syntax, 138
trace, 147
cleartext, 123
communication action, 40
completion, 61
knowledge, 75
standard form, 66
component, 36
computational complexity, 82
cryptogram, 123
decryption, 124
digital signatures, 124
encryption, 123
final state, 146, 167
index variable, 151
initial state, 146, 167
instance, 142
integrity, 128
interaction pattern
granularity, 83
join operator, 50
local choice, 42
mapping, 49, 101
  rule, 100
matching, 55
model checking, 18
nonce, 124
p-action, 100
principal, 125, 139
private key, 124
public key, 124
role, 97
secrecy, 128
security protocol, 123, 125
session, 48, 142
  acceptable, 61
  closed, 48
  may-be correct, 60
  open, 48
  semantics, 54
  successful, 59
  totally correct, 59
  trace, 56
session weak simulation, 69
silent action, 41
symbolic intruder, 130
symmetric cryptographic algorithms, 124
trace, 56
transition system, 11
variable
  bound, 43
  free, 43
  open, 43
Hofstadter’s Law: It always takes longer than you expect, even when you take into account Hofstadter’s Law.

Douglas R. Hofstadter

Gödel, Escher, Bach: an Eternal Golden Braid