Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

# A Logic and Tool for Local Reasoning about Security Protocols

Bernardo Parente Coutinho Fernandes Toninho
(aluno nº25858)

2º Semestre de 2008/09
20 de Julho de 2009

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

# A Logic and Tool for Local Reasoning about Security Protocols

Bernardo Parente Coutinho Fernandes Toninho (aluno nº25858)

Orientador: Prof. Dr. Luís Caires

*Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.*

2º Semestre de 2008/09
20 de Julho de 2009

# Acknowledgements

I'd first like to thank Prof. Luis Caires for his support and unwavering guidance through the several areas and techniques used in this dissertation. I thank Hugo Vieira for his tutorial session and explanations of the initial implementation of the Spatial Logic Model Checker tool, even though some time had passed since the code was actually written.

I'd like to thank my friend Mário Pires for some suggestions and hacks I used in the tool implementation and for the tireless bouncing off of ideas, some more insane then others. I'd also like to mention Prof. Frank Pfenning for taking the time to look over my sequent calculus and suggesting improvements in some of the rules. I give great thanks to Luísa Lourenço for proof-reading this rather long (and perhaps tedious) document quite efficiently and I mention Prof. Pedro Adão for taking the trouble of reading my thesis and for the very interesting and motivating thesis defense discussion and for his suggestions.

I also greatly thank Cátia Ferreira for her deepest kindness and for her incredible friendship.

Last but not least, I'd like to thank my family (including its canine members), but specially my parents, for their support and for their (non-academic yet fundamental) teachings throughout the whole of my life, which I suspect will endure in my memory longer then many of the academic ones.

On a broader note, I'd like to thank Prof. Edsger W. Dijkstra, whom I never met. His papers, lectures and letters simply never cease to reveal the spirit and mind of a true (computer) scientist and will always remain a great source of inspiration.

This work was supported by a CITI research grant.

# Resumo

Esta tese aborda o problema de desenvolver uma lógica e as técnicas de *model-checking* associadas de forma a verificar propriedades de segurança, e a sua integração na ferramenta Spatial Logic Model Checker (SLMC).

Nas áreas de desenho e análise de sistemas distribuídos, existe bastante trabalho relacionado com a verificação da correcção de propriedades de sistemas. Neste âmbito, grande parte do trabalho orientado para a verificação de propriedades de segurança recorre a métodos precisos mas informais.

Neste trabalho, apresentamos uma simplificação do cálculo de processos Applied $\pi$-calculus, introduzido por Abadi and Fournet, para o estudo de protocolos de segurança. Seguidamente, desenvolvemos uma lógica espacial para o cálculo, extendida com modalidades relacionadas com conhecimento, com o objectivo de analisar protocolos de segurança utilizando o conceito de conhecimento local de processos.

Adicionalmente, mostramos que as extensões são consistentes e completas em relação à semântica pretendida, não tornando a lógica indecidível, assumindo algumas hipóteses razoáveis. Apresentamos também um algoritmo de model-checking e a prova da sua completude para uma classe considerável de processos.

Finalmente, apresentamos uma implementação em OCaml do algoritmo, integrada na ferramenta SLMC, desenvolvida por Hugo Vieira e Luís Caires, produzindo assim a primeira ferramenta orientada para a análise de protocolos de segurança baseada numa lógica espacial.

**Palavras-chave:** Análise de Protocolos, Lógicas e Linguagens Formais, Lógicas Espaciais, Model-Checking

# Abstract

This thesis tackles the problem of developing a formal logic and associated model-checking techniques to verify security properties, and its integration in the Spatial Logic Model Checker (SLMC) tool. In the areas of distributed system design and analysis, there exists a substantial amount of work related to the verification of correctness properties of systems, in which the work aimed at the verification of security properties mostly relies on precise yet informal methods of reasoning.

This work follows a line of research that applies formal methodologies to the verification of security properties in distributed systems, using formal tools originally developed for the study of concurrent and distributed systems in general. Over the years, several authors have proposed spatial logics for local and compositional reasoning about algebraic models of distributed systems known as process calculi.

In this work, we present a simplification of a process calculus known as the Applied $\pi$-calculus, introduced by Abadi and Fournet, designed for the study of security protocols. We then develop a spatial logic for this calculus, extended with knowledge modalities, aimed at reasoning about security protocols using the concept of local knowledge of processes.

Furthermore, we conclude that the extensions are sound and complete regarding their intended semantics and that they preserve decidability, under reasonable assumptions. We also present a model-checking algorithm and the proof of its completeness for a large class of processes.

Finally, we present an OCaml implementation of the algorithm, integrated in the Spatial Logic Model Checker tool, developed by Hugo Vieira and Luis Caires, thus producing the first tool for security protocol analysis that employs spatial logics.

**Keywords:** Protocol Analysis, Mathematical Logic and Formal Languages, Spatial Logics, Model-Checking

# Contents

# List of Figures

# 1. Introduction

With the advent of the Internet, there has been a widespread dissemination of distributed systems that take advantage of the flexibility and ubiquity of the Net to provide services to users such as *e-commerce*, home banking, etc. However, the ubiquity and distribution that these systems require in order to fulfil their objectives introduces new problems. Several problems arise from the absence of a global state, which limits the information that the several components of a system can have about each other (e.g. it is impossible to distinguish a component failure from a network failure). Another great class of problems of distributed systems is related to the security of the systems themselves.

Since potentially anyone can access the Internet, all communications that take place over this medium are inherently insecure in the sense that they can be intercepted and produced by anyone. Therefore, if a system deals with sensitive information (such as financial systems) it is paramount that it be robust enough to prevent the openness of the medium from interfering negatively with the system's operation. The research problem therefore lies in finding techniques that allow the development of systems in such a way that they are protected against these (and other) threats.

In distributed systems, security encompasses a great variety of fields such as access control, auditing and security protocols. A security protocol consists of a set of actions that aim at establishing some security properties, inevitably using cryptographic techniques [58]. In protocol research, there are two naturally arising categories. One focuses on cryptography: drawing on information theory and number theory, for instance, to develop more sophisticated and elaborate crypto-techniques [37]; the other focuses on the protocols themselves [25]: how (and if) the message sequences work at achieving the desired properties and what practices can aid protocol engineering and design.

When focusing on protocols as a means to achieving security properties, there exist several approaches to the problem of analysing security protocols, with varying degrees of formalism. On the formal landscape, there exist many different ways of tackling this problem, and several have proved to be successful in validating or disproving security protocols [42, 56, 57]. Some techniques employ complexity-theoretic artefacts, others use formalisms crafted specifically for the problem and some are based on algebras and logics. Regardless, all these methods share the common goal of verifying if a protocol is correct in respect to its desired properties.

When considering the techniques based on process algebras we can distinguish at least two distinct types of approaches: one deals with a high-level specification of the protocol, with which its easier to observe the required properties, and then establishes an equivalence between the specification and the abstract representation of the protocol. The other, which this thesis focuses on, uses the same abstract representation (model) of the protocol but instead of proving equivalencies with specifications, checks that the desired properties (expressed through logical formulas) hold in the model.

In this context, there exist a considerable number of process calculi and logics that are able to model and reason about interesting properties of distributed systems such as local deadlock freedom and safety in resource usage. A particularly interesting class of logics that have been proposed are spatial logics. These logics take advantage of the fact that many of the interesting properties of distributed systems can be interpreted in spatial terms. For example, *connectivity*, which states that there is always an access route between two different sites, can be viewed as a property of the spatial arrangement of systems. While the more traditional logics for concurrent and distributed systems deal with notions of behavior, spatial logics aim to characterize the spatial and temporal dynamics of systems, being able to separate and reason about different sub-components of a complex system.

When dealing with security protocol verification however, different types properties arise. For instance, its essential that *secrecy* (the critical information exchange performed in a protocol cannot be obtained by attackers), among other properties, holds in a protocol that seeks to establish secure communication channels.

## 1.1 Motivation

The underlying motivation for this thesis is that many interesting properties of security protocols can be interpreted as the possibility (or impossibility) of a participant (or attacker) of the protocol to obtain a certain piece of information, that is, the capability of some sub-component of a system to *know* a piece of information. In fact, security protocols are fundamentally about ways of limiting knowledge (of attackers) and augmenting or certifying knowledge (of principals). Therefore, since both attackers and principals can be interpreted as sub-components of larger systems, the combination of knowledge reasoning with the key notions of spatial logics (local reasoning) follows naturally as a way of verifying security protocols. While the concept of reasoning about knowledge isn't necessarily new in this field, the usual approach deals with observable (and global) or exposed knowledge, which provides a much coarser analysis when compared to local knowledge reasoning. Also, most of the existing logics in this area that deal with knowledge are too separated from the underlying operational model, which can result in an increased difficulty to accurately represent properties that mix knowledge with (high-level) operational details of protocols.

In this new framework, secrecy can be viewed as the impossibility of an attacker, when paired with a system running the protocol, to know the pieces of information that are required to be secret in the protocol, such as keys. Another interesting set of properties that can be expressed in terms of this notion of local knowledge is protocol correctness in regard to the proposed “goal" of the protocol. For instance, in a key distribution protocol, the protocol is correct if it ensures that the participants (which can be viewed as sub-components of a system that implements such a protocol) eventually know the distributed keys (and maintains secrecy of the keys).

## 1.2 Goals

The goal of this work is twofold: we seek to devise a logic that by combining the standard spatial connectives with knowledge (epistemic) modalities, allows local reasoning about information (both obtainable and constructible) of security protocols. It is also our intent to develop a model checking algorithm for the logic, in order to realise a tool for security protocol analysis.

Our logic operates over models represented by a process calculus. Therefore, we aim at developing a calculus with which to represent security protocols, by drawing inspiration from other established calculi with similar intent. This formal representation has enough expressiveness to model security protocols by modeling the principals of the protocol as processes with the ability to exchange messages, composed by names or applications of functions to names (to model standard security techniques such as encryption or hashing).

The modalities allow us to write assertions about the information a process holds, by inspecting what values are present and what values can be obtained by applying functions to the messages (this can model the use of keys to decrypt cyphered messages). For instance, if a process has a cyphered message and the key used in the cypher, its possible to assert that the process also has (knows) the contents of the message.

Our logic hence aims to provide a detailed analysis of security protocols and their correctness, since not only it allows reasoning about exposed knowledge (by reasoning about the exchanged messages), but also of the flow of information inside a closed system (with the aid of the spatial connectives).

Besides the development of the logic and a model-checking algorithm, we also develop correctness proofs for our algorithmic approach. Finally, we extended the Spatial Logic Model Checker tool (a model-checker for spatial logics) with the knowledge modalities developed for the logic, in order to provide a proof of concept tool that uses this framework for protocol verification.

## 1.3 Contributions

The contributions of this work are as follows:

- We developed a process calculus model for security protocols, inspired in existing $\pi$-calculi. We augmented the standard $\pi$-calculus with structured terms that can be used in communications, in order to model cryptographic operations. We also added a special output prefix that models non-deterministic attacker outputs.
- We introduced a spatial-behavioural-epistemic logic, oriented towards security protocol analysis, with precisely defined syntax and semantics. Such a threefold approach is novel in the field, seeing as other related work in this area focuses solely on epistemic or spatial approaches.
- We developed a formal theory of knowledge deduction for our $\pi$-calculus models, realised through a sequent calculus formulation.

- We obtained soundness, completeness and decidability results for our formal theory, which ultimately result in the decidability of the logic itself.
- We realised an implementation of a model-checking algorithm for the logic as an extension to the SLMC tool, thus producing the first tool aimed at security protocol analysis using spatial logic model checking.

We validate our approach with our formal correctness results and with analyses of examples, performed using our proof of concept tool.

## 1.4 Document Structure

This document is structured as follows: Chapter 2 describes in detail the context in which this thesis is inserted, going from the broader notions of security and distributed system analysis, to more concrete work on process calculi and logics, having the security protocol analysis "horizon" present along the presentation and throughout the examples.

Chapter 3 describes in detail the work developed in this dissertation, clarifying the motivations and proceeding to present the model and logic developed in this dissertation, each accompanied with explanations and running examples. In Section 3.3 we present our sequent calculus formulation of a proof system for knowledge formulas and the theoretical results that aim at validating this work.

Chapter 4 details our extensions to the SLMC tool, by first giving some insight on the syntax and implementation of the original version and then going into our extensions *per se*. We then present two examples of protocols analysed using our tool, first a simple toy protocol and then the Needham-Schroeder protocol, as a way of validating our extensions and showing how the tool can be used to reason about protocols.

# 2 . An Overview of Security Analysis

This chapter presents the areas and body of work in which this thesis is inserted. Section 2.1 gives a broad view on the aspects of security, its applications, the need for protocol verification and some of the methodologies used in these analyses. Section 2.2 starts by giving a brief introduction on the techniques used to study concurrency theory and introduces the unified view of concurrency theory and security analysis using process calculi. Sections 2.2.2 through 2.2.7 present several of the most relevant contributions to the fields of concurrency and distributed system analysis using formal methods, specifically with process calculi and process logics, showing the progress from the earlier process calculi and logics that focused specifically on concurrency, to the more modern and sophisticated frameworks that reason about distributed systems in general, and enable security protocol analysis using richer calculi and logics.

## 2.1 Security in Computer Science

The problem of security is present in almost every aspect of Computer Science. Be it in operating system design, where the system level processes must be separated and secured from the user level ones, or in distributed systems, where complex security requirements such as authenticity of principals and message integrity are required, security is a concern of many fields in CS.

Despite being an aspect that touches many fields, the area in which security has had most of its application (and the area that generated most advances to security in general) is the area of distributed systems [27, 31, 61, 39, 41, 26]. While other areas have some security requirements, none have such a strong need for security as the distributed systems field of work. In this area, due to the uncertainty that is inherent to distributed systems, because of the absence of global state for example, security appears as a way of providing more information about this uncertain state, such as the exact identity of the emitter of a message, or a guarantee that only a certain number of principals can obtain some piece of information.

Realistically, distributed systems use a wide spectrum of tools developed in the field of security because they are required to, due to the open medium over which communications take place (usually the Internet). Since, in theory, anyone could potentially intercept all information exchanged in a distributed system, security guarantees are required to protect sensitive information and to protect the systems against malicious users. For example, home-banking or e-commerce systems both provide services that deal with information that, if divulged, would have dire consequences to its users. Therefore, such systems require that the communications with clients respect certain properties in order to assure that no harm comes from their usage. For example, these systems commonly need to be certain that the entity they are communicating with is in fact a valid client, or require that all communications with the clients are such that no eavesdropper can use them in a malicious way, or often a combination of both these requirements. In essence, security in distributed systems appears as a natural necessity due to the nature of the medium used to support communication and information exchange, to circumvent malicious use of information and to control and monitor its access.

### 2.1.1 Applications of Security

Security in distributed systems focuses on specific properties which appear naturally as security requirements. These security properties, although being requirements of concrete systems, exist at a high level of abstraction, since they are general enough to be of interest to any system with security requirements. As mentioned previously, systems often require that communications are held secretly, in order to protect their contents. This property is known as *confidentiality*, or *secrecy*, and is standard in practically all contexts where security techniques are applied. Another common property, *authenticity*, relates to making sure that the participants in communications are “who they say they are”. Communication *integrity* relates to being able to certify that exchanged messages are received exactly as they were emitted, that is, their data wasn’t tampered with in any way. The concepts of *authorisation* or *access control*, while of importance to security in general, are of a more concrete nature then the abstract notion of authentication, seeing as generally the authentication of a principal authorizes it’s access to generally inaccessible services or information.

Given some properties that must hold in a system, like authenticity and confidentiality, its necessary to devise a procedure that will establish the desired properties. This procedure is called a *security protocol*. Inevitably, security protocols employ cryptographic techniques to achieve said properties [58] and much work exists with the goal of improving the existing cryptographic schemes [51, 37, 50]. In fact, some sophisticated protocols such as the Transport Layer Security [27] protocol and the Secure Socket Layer [31] protocol combine several cryptosystems together to further strengthen its security guarantees. In some scenarios, protocols also enforce properties that are not specifically barriers against attacks but properties that focus on legal aspects such as non-repudiation [62] (assurance that a party cannot refute the validity of a message).

Most security protocols focus on establishing secure channels between parties with varying degrees of security: some focus specifically on message integrity and confidentiality [28], others on participant authentication [49], and several combine all these security properties [61, 39]. In general, security protocols use a variety of tools to accomplish their goals: cryptography to ensure confidentiality or integrity, timestamps (nonces) for freshness proofs [26], message sequence numbers, padding, among many other security artefacts.

### 2.1.2 Analysis and Verification of Security Protocols

Despite the technicalities that security protocols use, their goal is to establish some desired security properties. Due to their highly sensitive nature, it becomes fundamental that some sort of verification or analysis technique is employed, in order to assure that the properties are indeed always established, and that no attacker can interfere negatively with the protocol in some way.

A great number of successful analysis techniques actively employ formal methodologies to obtain the required analysis [10, 8, 38, 24, 57, 7].

Some formal approaches (for instance [10]), represent the protocol using message sequences and then reason if (or how) an attacker can exploit the messages exchanged in rounds of the protocol. However, the relation of some of these techniques to the operational details is perhaps too tenuous. Several classes of attackers are usually considered, with varying degrees of attacking power. Other techniques are more implementation oriented (such as [49]) which focus more on the implementational aspects then on the conceptual problems of protocols - being valid analysis techniques nonetheless.

Formal reasoning frameworks prove to be useful since they provide a degree of rigourousness that is deeply relevant due to the critical nature of security protocols, leading to an increase of the confidence of some protocols and the discovery of limitations and flaws in others. Another interesting point of formal techniques is that in some cases they enable the use of automated tools like theorem provers [52] or model-checkers [7] to assist the analysis process. Formal methodologies require the development of an abstract representation of the fundamental concepts of interest to the analysis, in this case of the relevant security artefacts. To achieve such an abstract view of protocols, formal methods require a *model* in which to focus their efforts. Ideally, this model (which encompasses the formal representation of protocols, cryptography, attackers, etc.) will be as complete as possible, in the sense that it should include as much relevant details as possible, all the while abstracting away details which are not relevant to the analysis of the protocol itself.

In the formal "world" of security there exist two fundamentally distinct models: the complexity-theoretic model and the symbolic model. In this work, we focus on the symbolic model of security due to its tractability. We will now approach the scope and limitations of the symbolic model, by motivating it against the complexity-theoretic one.

### 2.1.3 The Formal Symbolic Model: Scope and Limitations

Protocols are usually modelled as programs, written in some formal notation, or as the set of possible executions of the protocol. This representation is usually accurate, since it focuses on the relevant operational steps of the protocol such as the crypto operations it performs on data and the messages that are sent and received over a network. As mentioned, the model must also include a representation of attackers. Generally, attackers are seen as agents that can interfere with runs of the protocol at any point: they can interfere with several runs of the protocol, intercept messages on any communication channel and inject any message they can produce (or replay). This view of attackers, while very realistic, is hard to represent formally. Given that an attacker can inject any message it can produce, it must be considered as a non-deterministic agent, but without the ability to guess the keys or timestamps that are of relevance to the protocol. A solution to this problem that is used in the complexity-theoretic model is to represent the attacker as a probabilistic program, subject to some complexity bounds. Unfortunately, such an approach is very hard to use in practice.

The attacker model considered in the formal symbolic model stems from the work of Dolev and Yao [29]. This approach provides a simple solution to the guessing problem described earlier, by making every non-deterministic choice of keys and nonces generate a fresh value - one that by definition cannot have been used before. The formal representation of cryptographic operations usually relies on some assumptions that simplify the model (to a somewhat idealised level, as explained further below):

- Given a cypher key, anyone can produce a message encrypted with such a key.
- Given a cypher key and a message encrypted with that key, anyone can produce the decrypted values.
- Encrypted messages cannot be produced by agents who do not possess both the clear values and the keys.
- Neither the key nor the clear contents of a cyphered message can be derived from it without knowing the key.
- Decryptions with wrong keys result in evident failures.

This symbolic view of cryptography, despite being simplistic, is often effective because such an abstract view of the cryptographic operations is usually sufficient for the analyses. However, this simplification comes at a price: some types of attacks (such as bitwise shifts on keys or nonces, or pattern analysis in messages) are excluded by the model itself, making it impossible to analyse the occurrence of such attacks using these symbolic methodologies. This critique of the symbolic model is usually presented by the complexity-theoretic view of cryptography and security, that presents a much higher level of detail that takes into account complexity bounds on attackers and probabilities of analysing repeated patterns in cypher-texts, as mentioned previously.

Despite the limitations present in the symbolic model regarding its idealisations, the methodologies it provides can result in rich frameworks for the analysis of security protocols, successfully testing for the occurrence of several classes of attacks (for example [42] where Lowe describes an attack on the Needham-Schroeder protocol using a tool called FDR). Additionally, there are currently some research efforts with the objective of providing a rigourous justification for such an abstract treatment of cryptography, such as [4], where Abadi and Rogaway prove that the symbolic representation of cryptographic operations is sound when analysed in lower level computational models, at least with high probability and against attackers of bounded computational power (as those studied in the complexity-theoretic models), under reasonable hypotheses.

Seeing as this view of security is inserted in the context of distributed systems in general, we will now overview some of the work on formal models and analysis of concurrent and distributed systems that have been proposed thus far.

## 2.2 Concurrent and Distributed System Analysis

In parallel to the efforts developed for security protocol analysis, a great amount of work has appeared regarding the systematic analysis and study of concurrent and distributed systems in general. This work generally branches in two areas (that very often intertwine): the one that deals with the formal and systematic modelling of relevant concerns in concurrent and distributed computation, such as the notion of process, communication, mobility; and the one that deals with the formal reasoning about properties of said systems.

Many approaches exist in the mathematical modelling of concurrent computation. However, the greatest body of work in this area most likely lies in the paradigm of process calculi. A process calculus is essentially a formal modelling language of concurrent systems, providing an abstract operational description of interactions between a set of independent agents or processes, combined with algebraic laws that allow processes to be manipulated in an equational fashion, thus allowing formal reasoning about equivalencies between processes.

### 2.2.1 Bridging Concurrency and Security Protocol Analysis

Process calculi provide such a rich framework in which to study concurrent and distributed systems and their properties from an operational perspective, that they have begun to be used in the analysis of security protocols. The formal approach to security protocol analysis starts with a high-level representation of the protocol (with varying degrees of operational detail). Usually, this representation takes the form of a program-like description of the protocol, in which the relevant aspects of the protocol are represented (application of cryptographic operations, receiving and sending messages, etc.). As will be presented later, process calculi provide similar representations of concurrent and distributed systems. Additionally, since security protocols are usually applied to distributed systems, process calculi with enough expressive power to represent the operations required to model security protocols present themselves as a union point between the study of concurrency and distribution, and the analysis of security protocols, since a security protocol can essentially be viewed as an abstract distributed system.

We will now present the bulk of work in the area of process calculi and frameworks for reasoning about concurrent/distributed systems in general, moving towards the calculi and techniques developed for and used in security protocol analysis.

### 2.2.2 The $\pi$-calculus

Despite not being the first full fledged process calculus ever developed (that being Milner's Calculus of Communicating Systems - CCS [43]), the $\pi$-calculus is without a question the most influential and important of the foundational process calculi.

While CCS results in an interesting model of concurrency, its standard version lacks expressive power for many common features of concurrent systems such as divergent behaviour, value passing communication and dynamically rearrange-able configurations.

$$
\begin{array}{rcll}
P, Q & ::= & \mathbf{0} & \text{(Null process)} \\
 & \mid & P|Q & \text{(Parallel composition)} \\
 & \mid & (\nu n)P & \text{(Name restriction)} \\
 & \mid & \alpha.P & \text{(Action prefix)} \\
 & \mid & !P & \text{(Replication)} \\
 & & & \\
\alpha & ::= & m(x) & \text{(Input action)} \\
 & \mid & m\langle n\rangle & \text{(Output action)}
\end{array}
$$

Figure 2.1: The $\pi$-calculus Syntax

To address these concerns, Milner introduced the $\pi$-calculus [44], a further development of the work on CCS, with the design goal of producing a minimal language, yet expressive enough to, unlike CCS, model the full spectrum of concurrent computations.

The language itself is similar to CCS, although with a few subtle differences. The core notion of the $\pi$-calculus is that of *names*. A name in the calculus plays a dual role, representing both communication channels and variables. In fact, the minimality of the language results from this duality. In light of this binary notion of name, the core calculus allows (monadic) communication of a name through a channel (which is also a name), allows restriction of a name to the scope of a process, parallel composition of processes and replication of processes (potentially infinite copies of a process running in parallel). Name restriction can be viewed as fresh name creation, in the sense that a restricted name is such that no other equal name may exist. In Figure 2.1 we present the core syntax of the $\pi$-calculus. $P$ and $Q$ refer to processes, $n$ and $m$ are names and $x$ is a variable.

One of the most important features of the $\pi$-calculus is called *scope extrusion* of names. This relates to the capability of a process, given some name restricted in its scope, to communicate this name through a channel, thereby broadening the scope of the name to that of the receiving process. This feature models mobility of names, and therefore of channels, allowing for a system's configuration to change over time in the sense that channels can be created and exported to different components of the system.

Although being a minimal language, the $\pi$-calculus has been proved to be Turing-complete in [45], where Milner presents two encodings of the $\lambda$-calculus using the $\pi$-calculus. The encodings (one simulating the *call-by-value* reduction strategy and the other *call-by-name*) were done using a new formulation of the $\pi$-calculus inspired by Berry's and Boudol's work [6]. This result aims to show that the $\pi$-calculus is to concurrency as the $\lambda$-calculus is to functional computation, that is, the $\pi$-calculus can be viewed as a canonical (and complete) way of modelling arbitrary concurrent processes, in the sense that it possesses enough expressive power to encode any computable (concurrent or otherwise) computation. Another similar result is given in [54], where Sangiorgi shows that a version of the $\pi$-calculus where processes can be communicated through channels provides no greater expressive power than the standard version of the calculus.

Essentially, these results motivate an approach to the study and analysis of concurrent systems in general that is based on the $\pi$-calculus or on calculi that augment the $\pi$-calculus with some extra features of interest to the domain to be studied. Such is the approach chosen in this work.

Following the work on process equivalencies developed for CCS, several process equivalencies were introduced for the $\pi$-calculus, depending on the intended level of detail that the equivalence aims at capturing. A standard equivalence is structural congruence ($\equiv$), which despite being mostly used as a technical device to simplify the operational semantics of the calculus, relates processes that are structurally identical, allowing for an algebraic treatment of processes.

Perhaps a more interesting set of process equivalencies, that equate processes up to behaviour instead of up to structure, are the *early* and *late* bisimilarities. The underlying principal of a bisimulation relation is a relation that equates systems that behave in the same way, in the sense that both can simulate the actions of the other. The idea is that two processes are *bisimilar* if they cannot be distinguished in any way from each other by an external observer.

The notions of *early* and *late* stem from the fact that in $\pi$, processes can communicate names. In late bisimilarity, the intuition is that processes are equivalent if they behave the same way (in the simulation-sense) regardless of the names that are input, that is, the equated processes denote the same function of input names. Early bisimulation provides a simpler definition of bisimilarity in which the matching of actions depends on the name being communicated (the name *early* derives from the fact that the input names are instantiated earlier in the bisimulation). Both these definitions were presented in [47, 48].

The intent of these equivalencies is to capture indistinguishability of processes under different contexts and to give a (much desired from an analysis perspective) compositional flavour to the semantics of the process operators of the calculus. However, neither early nor late bisimilarity are congruences for all operators of the calculus. To address this problem, a third definition of bisimilarity, called *open* bisimilarity, was presented by Sangiorgi in [55].

Bisimilarities in the $\pi$-calculus present themselves as tools for the analysis and study of concurrent and distributed systems, with the added bonus that the expressivity of the $\pi$-calculus is much greater than that of previous calculi (like CCS). Tools for the automated verification of such equivalencies already exist, such as [59].

Although this thesis doesn't focus specifically on process equivalence techniques, their relevance arises from the fact that they are central to the characterisation results of logics for the $\pi$-calculus, in the sense that many logics aim to characterise specific process equivalencies, providing a higher level analysis tool for the same set of processes, while some intend to discriminate more processes then the standard bisimilarities, and therefore produce richer analyses, as will be seen throughout this report.

$$P(ch, \text{msg}) \triangleq ch\langle \text{msg}\rangle.ch(x).P(ch, \text{msg})$$

$$Q(ch, ack) \triangleq ch(y).ch\langle ack\rangle.Q(ch, ack)$$

$$R \triangleq (\nu ch)(P(ch, msg) \mid Q(ch, ack))$$

$$Alice \triangleq (\nu ch)pub\langle ch\rangle.ch(x).\mathbf{0}$$

$$Bob \triangleq pub(y).y\langle \text{data}\rangle.\mathbf{0}$$

$$Sys \triangleq Alice \mid Bob$$

Figure 2.2: Examples of $\pi$-calculus Processes

#### 2.2.2.1 **Brief Examples**

In Figure 2.2 we present two simple examples of $\pi$-calculus processes.

The system on the left hand side consists of three processes, $P$, $Q$ and $R$. Process $P$ sends a message *msg* over a channel $c$ and waits for a reply on the same channel, repeating this behavior indefinitely; process $Q$ behaves symmetrically to $P$ - receiving a message over a channel $c$ and sending an acknowledge over $c$. Process $R$ composes $P$ and $Q$ in parallel, generating the channel $c$ and restricting it to the scope of $P$ and $Q$.

The system on the right hand side consists of also three processes, *Alice*, *Bob* and $Sys$. *Alice* generates a fresh channel *ch* and sends it over the public channel *pub*, afterwards awaiting for an input on channel *ch*. *Bob* waits for an input on the public channel and will then output on the received name the name *data*. This example clearly illustrates the *scope extrusion* feature. *Alice* starts off with the fresh name *ch* restricted to its scope and then extrudes the scope of *ch* by sending it over the public channel.

Even in these simples examples we can observe a limitation of $\pi$-calculus from a security perspective: Processes $P$ and $Q$ are communicating over a channel that is inaccessible by any other process (namely by attackers), therefore any communications through $c$ are guaranteed to be secure, and as long as neither $P$ or $Q$ send the channel to any other process, one can argue that even authentication is guaranteed. On the right hand side example, after *Alice* successfully sends the new channel to *Bob* we can be sure that no other process can intercept communications over the channel, establishing secrecy and integrity of communications.

Such scenarios are unrealistic, in the sense that one cannot expect that two processes communicate through a channel that only they can access, at least not without some previously ran protocol that establishes the security of the channel over which to communicate. Hence, from a security protocol analysis perspective, modelling secure communication channels with restricted names is not realistic.

A naive attempt would be to use messages over an open channel to represent the establishment of the security of the channel, this approach however is fairly indirect. A more suitable approach would be to use an open channel, and use an encoding of encryption techniques to establish the security of the channel. Such an encoding in the $\pi$-calculus, however, is too intricate for practical use. Some of the calculi presented in future sections (2.2.3,2.2.4) will introduce ways to circumvent these issues.

$$
\begin{array}{lcll}
P, Q & ::= & \ldots & \\
 & \mid & \text{case } L \text{ of } \{n\}_K \text{ in } P & \text{(Decryption)} \\
 & & & \\
\alpha & ::= & \ldots & \\
 & \mid & m\langle\{n\}_K\rangle & \text{(Encrypted Output)}
\end{array}
$$

Figure 2.3: The Spi-calculus Cryptographic Extensions

$$
\begin{gathered}
A \triangleq c\langle\{M\}_K\rangle \\
B \triangleq c(x).\text{case } x \text{ of } \{y\}_K \text{ in } Op(y) \\
Sys \triangleq (\nu K)(A \mid B)
\end{gathered}
$$

Figure 2.4: Spi-calculus Example

### 2.2.3 Spi-calculus

While the $\pi$-calculus does indeed have great expressive power, its foundational and minimality goals often result in many features of distributed protocols being very hard to appropriately encode: such as encryption and decryption of data, or more generally, arbitrary operations on values.

The possibility of reasoning about distributed protocols that use such features, namely security protocols, using process calculi is an appealing prospect. Despite the number of notations that exist to specify security protocols, they are either too implementation orientated [49], not providing a precise way of reasoning about the protocol itself, or, in spite of being more formal [10] are too unrelated of the operational details of the protocol itself. Therefore, due to the somewhat high-level implementational nature of process descriptions in process calculi, expressing security protocols using process calculi would provide an interesting neutral ground.

As such, Abadi and Gordon introduced the Spi Calculus [3]. The Spi Calculus is an extension of the pi-calculus with cryptographic primitives (these primitives are shown in Figure 2.3), aimed at the description and analysis of security protocols. In essence, the calculus builds on [48, 47] by adding numbers and pairs to the possible values of the language, as well as the possibility to encrypt and decrypt such values and transmit them over channels. The encryption operation simply cyphers the value $n$ with key $K$, whilst the decryption operations takes a value $L$ and binds the decrypted contents to $n$ by performing a decryption using key $K$. Using this calculus we can model protocols such as the one in Fig. 2.4. In this protocol, principal $A$ encrypts the message $M$ with key $K$ and sends it over channel $c$. $B$ receives the message and performs a decryption with the appropriate key, proceeding to perform some operation $Op$ on the decrypted value $y$.

The fundamental aspect of this calculus is the usage of restriction and scope extrusion to represent fresh secrets or values and the sharing of secrets, respectively, in order to accurately model security protocols (this representation interestingly coincides with the Dolev-Yao model of [29]); and of observational equivalencies to model security and integrity properties, therefore focusing on an observational view of the protocol, related to what an attacker can observe (or know) from the running protocol as a whole.

As mentioned, Abadi and Gordon express security and integrity guarantees through observational equivalence. Essentially, if a run of a protocol with secret $D$ cannot be distinguished from a run of the protocol with secret data $D'$ (for any $D'$) - both runs are observationally equivalent - then D is kept safe by the protocol. The idea is that the "observer" is an implicit representation of an attacker that can interact with the protocol.

The spi calculus presents itself as a process calculus for reasoning about security protocols, using observational equivalence to model security and integrity properties. This approach therefore focuses (indirectly) on the knowledge obtainable by the environment (an arbitrary attacker), and it's ability to discriminate this knowledge in different runs of the protocol, keeping the representation of the attacker implicit in the model. Some limitations of this approach lie in the global view of the protocol, which may be interesting for security and integrity, but not as much in the compositional correctness of the protocol itself, since it doesn't allow for compositional reasoning. Also, the described observational equivalence isn't clearly defined and may be very hard to prove, since it considers arbitrary terms. A more technical yet important limitation resides on the limited set of cryptographic operations, which limits the scope of analysable protocols. Another possible limitation resides in the fact that while the attacker is mentioned as an arbitrary spi calculus process, it's implicit representation introduces a degree of heterogeneity in the model, since it treats attackers in a special way.

Despite its limitations, the spi calculus provides an interesting framework in which to study and reason about security protocols, using key features of the $\pi$-calculus (name restriction and scope extrusion), combined with operations on values relevant to the domain, and with observational equivalence techniques that represent security and integrity property verification, in respect to an interfering attacker. This thesis, although not focusing on process equivalence techniques, shares some of the spi calculus approach to protocol modelling (although closer to the calculus of section 2.2.4), in the sense that it also uses restriction to represent fresh values such as nonces, or secret values like keys.

Many of the ideas presented in the spi calculus have been used (adapted) in the cryptographic protocol verification tools *Proverif* by Blanchet [7] and more directly in *STA* by Boreale [9], where given a model of a protocol using a language similar to the spi calculus, its possible to express and verify properties in terms of traces generated by the protocol.

### 2.2.4 Applied $\pi$-calculus

The Spi-calculus, presented in the previous section, builds upon the $\pi$-calculus by extending it with cryptographic primitives, in order to provide a precise framework for reasoning about

$$\Sigma \triangleq \{\mathrm{enc}, \mathrm{dec}\}$$

$$E \triangleq \{\mathrm{dec}(\mathrm{enc}(x, y), y) = x\}$$

Figure 2.5: Equational Theory and Signature

security protocols. However, one of the downsides of [3] is that it provides a limited set of cryptographic operations. Essentially, for each choice of cryptographic primitives, both calculus and proof techniques need to be crafted, therefore limiting the kinds of protocols that can be expressed and analysed with the calculus.

In [2], Abadi and Fournet present the applied $\pi$-calculus. This calculus, typically used for reasoning about security, provides a general extension of the standard $\pi$-calculus by introducing functions, value passing and equations among terms, therefore bypassing the limitation of [3] in respect to the cryptographic primitives. Much like how the spi calculus allows for encrypted values to be transmitted over channels, the applied $\pi$-calculus allows for values constructed with names and functions to be transmitted. These functions are defined through signatures. A signature is a finite set of function symbols, that is then equipped with an equivalence relation on terms that is used to test for term equality. This equivalence relation can be generated from sets of equational axioms or rewrite rules. In essence, it is this feature of the applied $\pi$-calculus that gives it it's generic qualities, seeing as one can now imbue arbitrary functions and their semantics in the calculus. For instance, with the signature $\Sigma$ and equational theory $E$ from Figure 2.5, we define symmetric encryption (with function symbol *enc*) and decryption (function symbol *dec*). The semantic of decryption is defined by the equation, that states that decrypting a value with the correct key succeeds in producing the encrypted value.

Another novel construct of [2] is that of an *active substitution*, which is essentially a persisting substitution that any process can apply. This is used in the calculus to capture the partial knowledge an environment may have on values. For example, a process upon outputting a term through a channel "creates" an active substitution that expresses this information. A set of these substitutions is called a *frame*, and therefore captures a chunk of partial knowledge an environment may have. Interestingly, Abadi and Cortier show in [1] that deduction from frames (the problem of whether a term is deducible from the knowledge of a frame and the equational theory) is decidable for a class of equational theories identified as *subterm convergent*. This class of equational theories imposes some restrictions on the types of functions that can be defined, but are rich enough to express a great deal of functions used in security protocols. The result obtained in [1] is relevant because it validates the possibility of reasoning about security protocols using knowledge (exposed or otherwise).

In [3], observational equivalence is used as a tool to capture authenticity and secrecy properties. Following the intent of the development of a solid proof theory in which to reason about

$$P(k, \text{msg}) \triangleq \text{ let } x = \text{enc}(\text{msg},k) \text{ in } ch\langle x\rangle.ch(z).\text{let } y = \text{dec}(z,k) \text{ in } P(k, \text{msg})$$

$$Q(k, \text{ack}) \triangleq ch(y).\text{let } m = dec(y,k) \text{ in let } r = enc(\text{ack},k) \text{ in } ch\langle r\rangle.Q(k, \text{ack})$$

$$R \triangleq (\nu\, k)(P(k, \text{msg}) \mid Q(k, \text{ack}))$$

Figure 2.6: Example Using Encryption

(applied $\pi$) processes, Abadi and Fournet develop several equivalencies and proof techniques for the applied $\pi$-calculus, namely observational equivalence, which reflects indistinguishability in regard to actions; static equivalence, which reflects indistinguishability of substitutions, in the sense that the processes deal with equal terms under the equational theory; and (labeled) bisimilarity, which is defined as standard bisimilarities but requires static equivalence to hold (to embody the equalities of the equational theory), and proved to coincide with observational equivalence.

The applied $\pi$-calculus, given its generic nature and comprehensive equivalence and proof techniques, provides an almost foundational calculus in which to reason about security protocols, since it consists of a $\pi$-calculus that can represent (rather than encode) any operations that the protocol may require in an homogeneous fashion, all the while maintaining high-level and precise reasoning. These characteristics were the paramount source of inspiration to the calculus used in this thesis, seeing as the goal was to use a calculus expressive and rich enough to express any protocol. Therefore, choosing a calculus much like the applied $\pi$-calculus seems a natural choice.

#### 2.2.4.1 **A Brief Example Redux**

In figure 2.6, the example presented in section 2.2.2.1, Fig. 2.2 (left hand side) is reformulated using a language much like the applied $\pi$-calculus (we add the *let* declaration for readability). The idea is that instead of using a restricted channel, an open channel will be used. We will also employ symmetric cryptography, as defined by the signature and equational theory of Fig. 2.5.

In this case, encryption is used to ensure *secrecy* - since only $P$ and $Q$ know the key $k$, only they can decrypt the exchanged messages - and *authenticity*, in the sense that since only $P$ and $Q$ know the key $k$, then only they can generate messages encrypted with such a key. In such a simple example, it's fairly evident that both the properties hold. However, in more complex examples (which are usually the ones of interest), some proof techniques must be used: either using process equivalencies or logics.

$$\begin{array}{lcl} P \models T & \triangleq & \top \\ P \models \neg A & \triangleq & not\ P \models A \\ P \models A \wedge B & \triangleq & P \models A\ and\ P \models B \\ P \models \langle\alpha\rangle A & \triangleq & \exists Q. P \xrightarrow{\alpha} Q\ and\ Q \models A \end{array}$$

Figure 2.7: Hennesy Milner Logic

$$P \triangleq c\langle\rangle.v\langle\rangle.\mathbf{0}$$

$$P \models \langle c\rangle\langle v\rangle T$$

$$P \models \neg\langle v\rangle T$$

Figure 2.8: HML Example

### 2.2.5 Behavioural Logics

#### 2.2.5.1 Hennessy-Milner Logic

Together with CCS, Milner and Hennessy presented a logic known as Hennessy-Milner Logic (HML) [35]. HML is a modal logic that can be used to specify temporal properties of CCS processes. HML consists of the standard minimal boolean connectives together with a temporal modality (known as diamond). This temporal connective expresses the possibility of a process performing an action and resulting in a process that satisfies some HML formula. The syntax and semantics for HML are presented in Figure 2.7. The boolean connectives are interpreted standardly. Formulas with the diamond modality are satisfied by processes that can perform action $\alpha$ and whose continuations satisfy formula $A$.

In essence, HML is a behavioural logic, in the sense that it can reason about the behaviour (through actions) of CCS processes. For instance, as shown in Fig. 2.8, given a process $P$ that first outputs on channel $c$ and then outputs on channel $v$, we can define such behaviour with the first HML formula. This formula states that after performing the action over channel $c$, one of the possible resulting processes (in this case there is only one possibility) can perform an action on $v$. We can also state that $P$ cannot perform actions on $v$ from the start, which is done through the second formula.

A fundamental result of this work is that the logical equivalence induced by HML coincides with the notion of bisimulation presented earlier, that is, two processes satisfy the same set of HML formulas if, and only if, they are bisimilar. This characterisation result is of great importance from an analysis perspective, and provides much insight on the discriminating power of HML. In fact, logical characterisation results give a precise notion of the expressivity of logics, in terms of on what level they can distinguish processes.

Using HML we can express properties of process behaviour and then check that the formulas hold in a given CCS model. If that is the case, then we can be sure that in all processes that are bisimilar to it (that observably behave in the same way) the properties will also hold. This feature induces a verification pattern based on properties that can (or not) be combined with the implementation/specification verification pattern mentioned previously, that uses processes as a specification.

In specification-based analysis where the specification is expressed through a process, usually there exists a great deal of precision in the sense that only a single model exists for the given specification. However, verification is more often based on the necessity that certain high-level behavioural properties hold in a system. Therefore, a logical (higher level) view of the specification through a set of properties becomes very natural. One just needs to state the behavioural properties in HML formulas and then check if the formulas hold in the CCS process that models the system. This property based approach coincides with the one pursued in this thesis.

Additionally, due to the logical characterisation result mentioned earlier, one can perform these analyses on the specification of the system and assure that on all correct implementations of the system (i.e. implementations that are bisimilar to the specification), the properties will also hold. This analysis pattern allows a greater degree of flexibility in the ways of verifying system correctness.

#### 2.2.5.2 **Logics of Behaviour for the $\pi$-calculus**

As mentioned previously, a logical approach to specifications provides a more abstract and higher-level reasoning about process properties then that of process equivalencies. Such a logic characterisation of process behaviour also exists for the $\pi$-calculus, using modal logics which very frequently are variants of HML ([46] by Milner, Parrow and Walker), adapted for the $\pi$-calculus. In essence, the fundamental intent of these logics is to specify processes up to bisimilarity, that is, to have the logical equivalence (satisfaction of the exact same set of formulas) of processes coincide with bisimilarity, not only to bridge both approaches on process analysis (none providing a greater discriminating power), but also to provide a framework equivalent to that described for CCS.

Another interesting set of logics combine variants of HML and [46] with the modal $\mu$-calculus [40], which is a calculus of fixed point operators. These logics (such as [23], proposed by Dam) allow the expressing of properties using co-induction, which allows the description of a system by dividing it into simpler sub-systems. For example, liveness properties ("a system can always perform some operation") can be cleanly defined in this manner.

These logics, although having the same discriminatory power of bisimilarities, aim at providing a higher level view of properties that relate more directly to the behavioural aspect of systems, because specifications using logical properties generally have more models than those using process descriptions. This high level approach is one of the key aspects of this dissertation, that despite of using a process calculus model is more concerned with providing ways to express and verify properties of protocols.

### 2.2.6 Spatial Logics

Despite the expressive power of the behavioural logics presented thus far, the types of properties they can express are somewhat limited to the pure behaviour of systems. For instance, behavioural logics cannot reason about messages exchanged inside a closed system, since their observational power is limited to the external (observable) actions. Additionally, from a distributed system perspective, it's interesting to reason about properties related to resource usage in general, for example to state properties of *race freedom* - absence of simultaneous consumption of a resource in an uncontrolled way. Also, to verify the correctness of a distributed system, one has to consider the several (abstract or concrete) locations inside the system itself, instead of simply considering the interactions of the system with its environment. This lack of expressiveness has motivated the development of a new class of logics: dynamic spatial logics for concurrent and distributed system analysis.

Dynamic spatial logics [12], instead of limiting their observational power to the labelled action dynamics of systems (states as nodes in a labelled graph or tree where transitions take place over time), explore structures where the states not only evolve in time, but possess an internal spatial arrangement that can be accessed through the spatial connectives [16]. While many spatial structures can be conceived, the structure usually considered in the scope of distributed system analysis, using process calculi, is to interpret a system as a set of "threads" (this set is constructed by the static operators of the calculus such as parallel composition in [48, 47]) that can be bound together using name restriction.

A spatial behavioural logic, in addition to a set of behavioural modalities (usually similar to those of HML), possesses a set of spatial operators that relate to the static operators of the process calculus for which the logic is intended [19, 13, 14]. In the case of the $\pi$-calculus (the calculus that serves as a starting point for this thesis' work), the static operators usually considered are the parallel composition operator ($P \mid Q$), the null process $\mathbf{0}$ and name restriction $(\nu n)P$. The logical counterpart of these operators (as defined in [13, 14] by Caires and Cardelli) are the composition formula $A \mid B$, which holds in a process that can be separated into two processes in parallel, one satisfying $A$ and the other satisfying $B$; the void formula $\mathbf{0}$, that holds in the void process; and the hidden name quantifier $\mathbf{H}x.A$ that allows quantification over restricted names - such a formula holds in a process that has an hidden name such that the formula $A$ holds (instantiating the name variable $x$ in $A$ with the hidden name in the process). A core spatial logic with these connectives is presented in Figure 2.9 (we denote by $fn(P)$ the set of free names of process $P$). Notice that the semantics of the logic make use of the structural congruence relation $\equiv$ to inspect the structure of the models.

With our core spatial logic we can express interesting properties of the spatial dynamics of processes. For example, we can state that a process $P$ is composed of two non-void subprocesses (Fig. 2.10 - top) or we can state that a process is composed of two non-void subprocesses under a name restriction (Fig. 2.10 - bottom).

$$\begin{array}{lcl}
P \models T & \triangleq & \top \\
P \models \neg A & \triangleq & \mathit{not}\ P \models A \\
P \models \mathbf{0} & \triangleq & P \equiv \mathbf{0} \\
P \models A \mid B & \triangleq & P \equiv Q \mid R\ \mathit{and}\ Q \models A\ \mathit{and}\ R \models B \\
P \models \mathbf{H}x.A & \triangleq & P \equiv (\nu n)Q\ \mathit{and}\ Q \models A\{x \leftarrow n\} \\
P \models @n & \triangleq & n \in fn(P)
\end{array}$$

Figure 2.9: A Core Spatial Logic

$$P \models (\neg\mathbf{0} \mid \neg\mathbf{0})$$

$$P \models \mathbf{H}x.(\neg\mathbf{0} \mid \neg\mathbf{0})$$

Figure 2.10: Spatial Properties

The combination of the previously described spatial operators with a name occurrence predicate, recursion, propositional operators and behavioural modalities (like those of [23] by Dam) gives rise to what can be seen as a basic spatial behavioural logic expressive enough to reason about distributed systems in respect to properties that relate to resource usage or internal behaviour. Examples of such properties are presented in section 2.2.6.4, based on the running example presented previously.

#### 2.2.6.1 **Local Reasoning with Spatial Logics**

One of the most important aspects of (dynamic) spatial logics is that they enable local reasoning about complex systems. Instead of just approaching a system as a whole, one can also reason about a system by dividing it into smaller sub-components and reasoning separately about them, which is a much more interesting approach when analysing distributed systems (and protocols), since generally one is not only interested in the behaviour of the system in its entirety but also of the various parts that make up the system itself. Local reasoning is achieved by a combination of spatial connectives such as the parallel composition formula, the hidden name quantifier, a name occurrence predicate and fix-point recursion. The parallel composition formula separates various sub-components of the system to be analysed, while the hidden name quantifier, the name occurrence predicate and recursion can be combined to “open” a system, by lifting all restricted names and allowing reasoning about the internals of the system. This lifting is achieved by successively quantifying over the hidden names of the system, until no other hidden names can be quantified and therefore the system has been completely “opened”. One can then use the parallel composition formula to navigate the internal spatial substructure of the system. With this framework one can then write properties such as *unique handling*: by stating that internally, there never exists a configuration of the system where two of it's subsystems attempt to read simultaneously off the same channel; *race freedom*: by stating that there never exists an internal configuration of the system where more then one subsystem outputs on the same channel (that in turn is being read by some subsystem).

Spatial logics also allow one to count the number of components of a system by reasoning about the impossibility of dividing the system in a specific number of non-void components. This technique can be used to reason about more general properties of the components of a system: one can state that all components in the system satisfy some property, or that a certain set of components satisfies some property.

Another interesting feature of spatial logics is their adequacy in expressing contextual tests. In many process calculi, there exist notions of arbitrary contextual tests, since they provide a uniform and general approach to behavioural specifications and observational equivalencies - by stating the behaviour of processes i.r.t any context in which it can exist. In spatial logics there's a logical primitive that expresses the concept of contextual test, coined as the "guarantee" operator in [14]. Used as $A \triangleright B$, this formula holds in a process $P$ if, for every process $Q$ that satisfies $A$, the parallel composition of $P$ and $Q$ satisfies $B$.

To some extent, one can even reason about *secrecy* using classic spatial logics, in the sense that it is possible to state that a process will never leak private (restricted) names on public channels, by using hidden name quantification. However, one should note that the $\pi$-calculus isn't the best calculus for analysing systems or protocols designed for security, thus when using a more security-friendly calculus such as those discussed in sections 2.2.3 and 2.2.4, secrecy cannot be interpreted in such a naive way. In a scenario where a process holds some private name, it may send it through an open channel and not leak it if the output is *safely* encrypted, that is, it outputs a cyphered message that contains the private name and the cypher key is not known by undesired third parties. Also, it may be possible to output a private name by encasing it in an encrypted message that can be deciphered by attackers sometime in the future, that is, the attacker eventually obtains the key that was used to cypher the initial message.

This argument shows that while dynamic spatial logics (in this case for the $\pi$-calculus) indeed provide a powerful framework for reasoning about distributed systems in general, they are somewhat ill-equipped to realistically reason about security since they cannot (and were not intended to) reason about composite terms such as cyphered messages, which can encapsulate and expose information depending on the information held by the receiver of a message. Therefore, some extensions need be made to address security properties, when dealing with security-oriented calculi such as [3] or [2]. In [30], Lozes and Villard present a spatial equational logic designed for the applied $\pi$-calculus. The logic is an extension of first order equational logic with spatial connectives. The underlying idea is that term equality is defined from both the equational theory under consideration and local axioms from applied $\pi$-calculus frames. This logic, while having similar goals as the one in this thesis, does not deal with the concept of "knowledge" of a process, therefore being a somewhat alternate approach to fundamentally the same problem.

While the discussion of spatial logics thus far has only presented the hidden name quantifier and a name occurrence predicate as a way of reasoning about restricted names, there exist alternative spatial operators able to express such properties, as proposed in [19] by Cardelli and Gordon. Such operators are the revelation operator, written as $x®A$, that holds in processes that have a restricted name $x$ and it's underlying process satisfies $A$; and the freshness quantifier $\text{И}x.A$, which quantifies over fresh names. Interestingly, the hidden name quantification can be expressed by combining the freshness quantifier with the revelation operator, and the name occurrence predicate can be expressed by the revelation operator alone. Conversely, the freshness quantifier and revelation operators can be expressed using name occurrence and hidden name quantification. Its even possible to define validity and entailment within the logic itself using the guarantee operator [19]. In [13, 14], Caires and Cardelli develop in great detail the proof theory and meta-theoretic properties of a spatial logic for the $\pi$-calculus, including a sequent calculus formulation of a (sound) proof system for the logic.

### 2.2.6.2 **Expressiveness and Decidability**

The aim of spatial logics is to give a greater degree of expressiveness than classical behavioural logics, essentially allowing to analyse processes in more detail. There exists much work regarding the expressiveness of spatial logics, in the sense of what exactly is their separation power for the model which they are designed for, or what equivalence on processes does the logical equivalence induce.

In [11], Caires concludes that the separating power of a dynamic spatial logic (with action modalities, fix-point iteration, fresh quantification and revelation) lies between structural congruence and strong bisimilarity, that is, its finer than bisimilarity and coarser than structural congruence. In fact, a logic characterisation of bisimilarity in the finite $\pi$-calculus was developed in [36] by Hirschkoff, using a spatial logic without parallel composition but with name occurrence and the guarantee operator, used as a contextual testing primitive. Other characterisation results of spatial logics have been developed, such as [17] by Caires and Vieira, regarding extensional observational equivalencies.

Relevant to the study of the expressiveness of the logics is the study of the decidability of model and validity checking. Given the expressiveness of spatial logics, it's expected that many of the decidability results come out negative, as shown in [21] by Charatonik and Talbot, where both model-checking and validity of pure spatial logics (without behavioural modalities and with the guarantee operator and universal quantifiers over names) are proved undecidable. However, Calcagno, Cardelli and Gordon show in [18] that the static fragment of spatial logics (the void, parallel and guarantee operators) is decidable, although with a very high complexity. In an important result by Ghelli and Conforti [33], it is shown that the source of undecidability is the revelation operator, while the static spatial operators with the freshness quantifier remain decidable. Following [33], it was proved by Caires and Lozes that dynamic spatial logics, in the presence of the guarantee operator (which adds to the expressiveness of the logic to the extent that it cannot be removed), are essentially undecidable [15].

As described, spatial logics, given their high degree of expressiveness, are undecidable and incomplete in general. However, spatial behavioural logics without the guarantee operator remain very expressive and useful from a verification point of view (the presentation in section 2.2.6.1 for instance), and it's shown in [11] that the model-checking problem is decidable and complete for bounded processes, which includes the class of processes with finite control (finite state space).

### 2.2.6.3 **The Spatial Logic Model Checker**

The initial version of the Spatial Logic Model Checker [60] consists of a model-checking tool for a spatial behavioural logic [11] for the $\pi$-calculus. Its algorithm is both sound and complete for bounded processes (a process whose set of reachable processes after an arbitrary sequence of spatial/behavioural observations is finite). The syntax of the tool is similar to that of the $\pi$-calculus and of standard spatial-behavioural logics, although with some syntactic enrichments to provide a simpler use.

In this thesis, the tool has been extended with support for term communication (terms are values built from application of symbolic functions to names), and the ability to reason about the knowledge of a process. In this context, knowledge is interpreted as the set of terms a process holds, as well as those it can derive from them.

### 2.2.6.4 **An Example with Spatial Logics**

Considering the initial (and very simple) example of Figure 2.2, it's interesting to see what kinds of spatial properties one can write, despite their triviality. For instance, we can state that the process $R$ is composed of two distinct sub-components under a hidden name:

$$R \models \mathrm{H}x.(\,\mathbf{1} \mid \mathbf{1}\,)$$

With:

$$\mathbf{1} \triangleq \neg\mathbf{0} \wedge \neg(\neg\mathbf{0} \mid \neg\mathbf{0})$$

We can also state that $R$ is composed of two sub-components, one that has a name *msg* and sends it to the other (which can be expressed by the HML behavioural modality $\langle x\rangle\phi$ - "its possible to perform action $x$ and after that $\phi$ holds"), and the other that has a name *ack* and sends it to the other:

$$R \models \mathrm{H}x.(\,\mathbf{1} \wedge @msg \wedge \langle x\langle msg\rangle\rangle\top \mid \mathbf{1} \wedge @ack \wedge \langle x()\rangle\langle x\langle ack\rangle\rangle\top\,)$$

And more generally, we may want to state that $R$ never opens the private communication channel. This notion of invariant property is embodied by the $\Box\phi$ connective that combines fix point iteration with the dual of diamond - "in every reachable state, $\phi$ holds":

$$NoLeak \triangleq \Box\neg\exists p.\mathrm{H}x.\langle p\langle x\rangle\rangle\top$$

$$R \models NoLeak$$

These example properties show some of the expressive power of dynamic spatial logics. Although, considering the modified example of Figure 2.6, where instead of using a private channel, communications take place over a public channel using cyphered messages, the *NoLeak* property wouldn't have its intended meaning, since one could easily conceive an example where *NoLeak* holds, yet a private name is still leaked:

$$S \triangleq ((\nu pvt)\ \text{let}\ x = f(pvt)\ \text{in}\ ch\langle x\rangle) \mid ch(y).\text{let}\ z = f^{-1}(y)\ \text{in}\ \ldots$$

The process on the left-hand side of the composition has a private name *pvt* and outputs the result of applying some function $f$ to *pvt* over a channel. The right-hand side process simply inputs over the channel and applies the reverse function $f^{-1}$ on the received message, therefore obtaining *pvt*. Despite this, formula *NoLeak* holds in $S$, since it never directly outputs *pvt* over the channel. This counter-example clearly shows that great care is needed when considering spatial logics for process calculi that allow actions like function application.

As previously mentioned in Section 2.2.6.3, in the development of this thesis the SLMC tool has been modified at the model and the logic level, however, we present in Fig. 2.11 the original example of Figure 2.2 using the SLMC syntax without any of the new features. The SLMC tool provides a few syntactical facilities to ease the writing of properties, such as the *inside* predicate, which lifts all restricted names of a process; the **1** predicate, which holds if a process is composed of a single "thread" (the SLMC implements a more generic $k$ predicate, which holds in a process that is composed of $k$ threads); the *always A* predicate, which is an implementation of the $\Box$ modal operator combined with fix-point iteration, which holds in a system where $A$ is true in every possible step; and the *eventually A* predicate, which holds in a system where $A$ is true in a possible step of its execution.

In the process definitions, the *new* construct defines restricted names; an output through a channel is written *channel!(*$x_1,\ldots,x_n$*)*, with $x_1,\ldots,x_n$ being the names to be output ; and inputs are written *channel?(*$x_1,\ldots,x_n$*)*, with $x_1,\ldots,x_n$ being the variables over which the input will be read.

### 2.2.7 Epistemic Logics

The several logics presented up to this point deal specifically with notions of behaviour and spatial dynamics of their intended models. However, some security properties aren't naturally expressed solely as a function of the behaviour or the spatial dynamics of principals, as argued in Section 2.2.6.4. As such, there has been an effort to use different types of logics to reason about security properties in a more direct fashion. Specifically, epistemic logics, or logics of knowledge and belief, have been used to reason about security since many security properties can be understood as a function of the knowledge, or beliefs, of the principals involved in a protocol.

The idea of using knowledge follows naturally from the fact that fundamental properties of security are essentially properties of knowledge. Authenticity, for instance, can be understood

```
defproc P(ch) = ch!(msg).ch?(x);;

defproc Q(ch) = ch?(y).ch!(ack);;

defproc R = new c in (P(c) | Q(c));;

defprop noLeak = always not exists p.hidden x.<p!(x)> true;;

defprop twoComp = inside (1 | 1);;

defprop behaviorR = hidden x.(
          (1 and @msg and <x!(msg)> true) |
          (1 and @ack and <x?><x!(ack)> true));;

check R |= noLeak;;
check R |= twoComp;;
check R |= behaviorR;;
```

Figure 2.11: Example for the SLMC

as knowing the sender of a message, anonymity can be read as not knowing the sender of a message, among other properties.

With this view of security in mind, BAN logic [10] was introduced. This logic, although lacking clearly and precisely defined semantics, deals in assertions of the belief of agents in a protocol. Another potential flaw of BAN (or other purely epistemic approaches [22]) is that modelling protocols using epistemic-based approaches is a non-trivial feat, essentially because logics are geared for specifying properties instead of operational steps of protocols. Considering this weakness of more classically based epistemic frameworks, several authors have proposed hybrid frameworks that model protocols using process calculi and reason about properties using epistemic modalities.

In [24], a temporal-epistemic logic is introduced over a CCS-like process calculus, with sequential and parallel composition, but with an epistemic flavour where actions can be tagged with the identities of the principals who observe them. The logic itself is a variant of the $\mu$-calculus with epistemic modalities and past. In this logic, the standard "knows" modality is instantiated with the identity of the agent whose knowledge one wishes to verify. For instance, to state that agent $i$ knows $\varphi$ we would state $K_i\varphi$, which would essentially range over all states considered possible by agent $i$ - hence the action tagging of the calculus. This framework, although much closer to a pure operational approach, still has the issue of the calculus having explicit information over which agents observe what actions, which from a purely operational perspective isn't a very natural interpretation of the operational definition of a protocol.

In a very recent development however, [20] introduces an epistemic logic for the applied $\pi$-calculus of Section 2.2.4, without replication. This logic has some similarities to the one developed in this thesis, in the sense that it has modalities to reason about terms. These modalities, however, are fundamentally different. The logic is separated in three types of formulas: term denotations, static formulas and epistemic formulas. Term denotations are logical representation of applied-$\pi$ terms; static formulas consist of first order quantification, disjunction, negation and a predicate that can be used to reason about the set of terms the intruder possesses; finally, the epistemic formulas consist of past and future operators, and an epistemic knowledge (from the intruder point of view) operator. The epistemic component of the logic is interpreted over the set of all possible runs of a protocol, using a notion of maximal trace of a process - which coincides with all possible runs because the model lacks replication. This approach is essentially based in the intruder point of view, and interprets epistemic properties as properties of all the runs of a protocol (therefore being limited to finite traces).

Its interesting to see the contrast of this framework with the one presented in this thesis, since both are introduced at similar dates. Here we focus on static knowledge from the perspective of the principal instead of the attacker (allowing reasoning about the internal behaviour of the protocol at the "expense" of explicit attacker modelling), and combine this view of knowledge with spatial and behavioural modalities, therefore allowing compositional reasoning about the knowledge of the several principals (and attackers), over time and space. These differences result in fundamentally different work from that of [20], in the sense that their somewhat opposite perspective requires an equally opposite formal treatment, closely related to static equivalence of processes and of traces.

# 3 . A Logic for Local Knowledge Reasoning

In this chapter we present the bulk of the work developed during the course of this thesis. We begin by motivating the model that underlies our logic, followed by its syntax and some necessary term theory, as well as the calculus' formal semantics.

We then step into our logic, presenting the syntax and denotational semantics for the core spatial-behavioural logic extended with the new term (epistemic) modalities. We follow by presenting some useful derived idioms, and we present some ways of stating some interesting properties from a security perspective.

Following the logic, we present our theory of the computational capabilities of principals and attackers in a sequent calculus formulation and present some important theoretical results that aim to validate the approach. We begin by showing that the sequent calculus formulation is sound in respect to the ability to derive new information from terms (through reduction and function application), present in the semantics of the logic. We then present a sound, complete and decidable procedure of calculating an approximation of the full (infinite) derivable knowledge of a process and we prove the sequent calculus to be complete in respect to this approximation. Through this result we are then able to prove the completeness of the sequent calculus in respect to the full derivable knowledge of processes. Thus proving our formal theory to be both sound and complete.

Finally, we present the extensions implemented in the SLMC tool and show how the previous proofs provide a way to algorithmically determine if a term (or set of terms) can be derived from an initial set of terms.

## 3.1 Model

As mentioned in previous sections, we need a model that, while preserving abstraction, is expressive enough to model security protocols with a high-level of operational detail, since that is the most straightforward way of modeling a protocol. As argued in Section 2.2.2.1, despite the Turing completeness of the $\pi$-calculus, encoding cryptographic operations in it would be cumbersome and too complex, therefore we follow an approach that is somewhat a hybrid between the Spi-calculus [3] and the Applied $\pi$-calculus. The idea is that we want a calculus that has the expressiveness of $\pi$ but where we can easily model and use cryptographic operations such as symmetric and asymmetric encryption and hashing, in order to model the several principals and adversaries of a protocol as processes in the calculus. We therefore take the $\pi$-calculus and extend it with the capacity to communicate arbitrary structured terms (built from the application of function symbols to names), defined by a term algebra. These terms symbolically model the necessary cryptographic operations. We refrain from using some of the technical aspects (such as frames) of the Applied $\pi$-calculus due to simplicity, and because we aren't exploring the equivalencies induced by the semantics of the calculus. To define the semantics of cryptography, we use a signature, which is a set of function symbols equipped with an arity, and a set of equations (or rewrite rules) that describe how we can perform operations on terms. Following [1], we impose some restrictions on the form of the equations that can be used, to ensure decidability.

We now present the syntax of our process calculus. In the construct for recursion we require all occurrences of process variables to be guarded by an action. In the choice construct we require both choice components to also be guarded by an action.

### 3.1.1 Syntax

**Definition 3.1.1.** ***(Processes).*** *Given a set of function symbols $\Sigma$ called a signature, an infinite set of names $\Lambda$ such that $m \in \Lambda$ and $n \in \Lambda$, a set $\chi$ of process variables $X$, the set of processes $(P, Q)$, of actions $\alpha$, of terms $M$ are defined by:*

$$
\begin{array}{lcll}
P, Q & ::= & \mathbf{0} & \textit{(Null process)} \\
 & | & P|Q & \textit{(Parallel composition)} \\
 & | & (\nu n)P & \textit{(Name restriction)} \\
 & | & \alpha.P & \textit{(Action prefix)} \\
 & | & P + Q & \textit{(Choice)} \\
 & | & \textit{let } n = M \textit{ in } P & \textit{(Let construct)} \\
 & | & (\mathbf{rec}\, X.P) & \textit{(Recursive definition)} \\
 & | & X & \textit{(Process variable)} \\
 \\
\alpha & ::= & m(x) & \textit{(Input action)} \\
 & | & m\langle M \rangle & \textit{(Output action)} \\
 & | & [M_1 = M_2] & \textit{(Test)} \\
 & | & m\langle * \rangle & \textit{(Attacker output)} \\
 \\
M & ::= & n & \textit{(Name)} \\
 & | & f(M_1, \ldots, M_a) & \textit{(Function application)}
\end{array}
$$

*where the function symbol $f$ ranges over functions in $\Sigma$ and $a$ matches the arity of $f$.*

The syntax uses constructs commonly used in process calculi, in particular its very close to the syntax of the Spi-calculus and the Applied $\pi$-calculus. The key features of the calculus lie in the usage of structured terms as messages and explicit attacker representations. The latter is achieved through the Attacker output construct which is a non-deterministic output of any message that a process can know.

Since the usage of these terms intends to model cryptographic operations, we need to be able to define how to transform a term into another term. These transformations allow us to represent decryption, hashing, pairing, or other useful and necessary operations.

To achieve this, we define an equivalence relation $=_E$ on terms that represents semantic equality of terms. Such a relation can be generated from a set $E$ of equations. Each equation

$$E_1 \triangleq \{\mathrm{dec}(\mathrm{enc}(x,y),y) = x; \pi_1(\mathrm{pair}(x,y)) = x; \pi_2(\mathrm{pair}(x,y)) = y\}$$

$$E_2 \triangleq \{\mathrm{dec}(\mathrm{enc}(x,y),\mathrm{pk}(y)) = x; \mathrm{dec}(\mathrm{enc}(x,\mathrm{pk}(y)),y) = x\}$$

Figure 3.1: Equational Theories

has the form $M = N$ where $M$ and $N$ are terms. A finite set of equations is called an equational theory. We obtain $=_E$ by considering the symmetric, reflexive and transitive closure of $E$. Additionally, we impose a constraint on the form of equations we consider, by forcing each $N$ to be a subterm of $M$. Such equations may be oriented $M \rightarrow N$ such that $N$ is a subterm of $M$. The motivation for these syntactic restrictions follows from the result of [1], where deduction for such (convergent) subterm theories was proved decidable.

### 3.1.2 Equational Theories and Term Rewriting

While the previously mentioned restriction does indeed diminish the kind of operations we can model, we can still model an interesting set of operations, as show in Figure 3.1, where theory $E_1$ contains the equations for defining projections on pairs and symmetric cryptography, and theory $E_2$ contains the equations for asymmetric cryptography.

For further reference, we call the top level function symbol on the left-hand side of an equation a *destructor* in the sense that it "destroys" the inner terms to produce the smaller term on the right-hand side of the equation. Other function symbols are called *constructors*, since they allow us to build up new values from names and other terms. For instance, when considering the equation $\mathrm{dec}(\mathrm{enc}(x,y),\mathrm{pk}(y)) = x$ from theory $E_2$ we have that the function dec is a destructor and the functions enc and pk are constructors. We denote constructor with the symbols $f,g,h$ and destructors with $\delta$.

Given an equational theory, we can assign directions to its equations such that for every equation $t_1 = t_2$ we have $t_1 \rightarrow t_2$. Such a rule is called a rewrite rule in the literature. We will now introduce some theory on term rewriting, in order to precisely define the notion of convergence for equational theories.

**Definition 3.1.2.** ***(Term Rewriting System).*** *Given a set of rewrite rules generated from an equational theory E, we call this set a term rewriting system. Generically, given a signature $\Sigma$, a rewrite rule over $\Sigma$ is a pair of terms $(s,t)$ built from functions in $\Sigma$, written $s \rightarrow t$, so that $s$ is not a variable and the variables of $t$ are a subset of the variables of $s$.*

If $R$ is a term rewriting system over $\Sigma$, for two terms $v_1, v_2$ we say that $v_2$ has been obtained by a one-step reduction from $v_1$ using $R$, written:

$$v_1 \rightarrow_R v_2$$

If for some rule $(s,t) \in R$, $v_1$ can be matched with $s$ and $v_2$ can be matched with $t$, by instantiating variables in the rule. We write $\rightarrow^*$ as the reflexive, transitive closure of one-step reduction, and we write $=_R$ as the reflexive transitive symmetric closure of one-step redution, and say that $s$ reduces to $t$ if $s \rightarrow^* t$. We call a term $t$ a normal form in $R$ if there is no term $s$ such that $t \rightarrow s$. Finally, we say that a term rewriting system $R$ is a term rewriting system for an equational theory $E$ if for all terms $s,t$:

$$s =_E t \leftrightarrow s =_R t$$

We also define a notion of name occurrence on terms, that is helpful in future definitions.

**Definition 3.1.3.** ***(Name Occurrence).*** *We denote by $names(t)$ the set of names (not function symbols) that occur in term $t$.*

We can now introduce the concepts of confluence and strong normalisation necessary for the convergence property of [1].

**Definition 3.1.4.** ***(Confluence).*** *A term rewriting system is* confluent *if for any terms $r, s_1, s_2$ with $r \rightarrow^* s_1$ and $r \rightarrow^* s_2$ there is a term $t$ such that $s_1 \rightarrow^* t$ and $s_2 \rightarrow^* t$. For such term rewriting systems, normal forms are unique if they exist, since if $s$ is a normal form and $s \rightarrow^* t$ then $s = t$.*

**Definition 3.1.5.** ***(Strong Normalisation).*** *A term rewriting system is* strongly normalizing *if there are no infinite sequences of terms such that:*

$$t_1 \rightarrow t_2 \rightarrow \ldots$$

**Definition 3.1.6.** ***(Convergence).*** *If a term rewriting system is both strongly normalizing and confluent, it is complete or convergent.*

For more on term rewriting theory, we refer to [5]. Before presenting the operational semantics of the calculus, we define a notion of term depth and functional equational closure of a set of terms, that will be necessary later on.

**Definition 3.1.7.** ***(Depth of a term).*** *We define the maximum nesting depth of a term $T$, represented as $|T|$, as follows:*

$$\begin{aligned} |n| &\triangleq 0 \\ |f(t_1,\ldots,t_n)| &\triangleq 1 + \max(|t_1|,\ldots,|t_n|) \end{aligned}$$

**Definition 3.1.8.** ***(Functional and Equational Closure).*** *Given a set of terms $S$ and a signature $\Sigma$, we define the functional equational closure of $S$, $\mathfrak{F}(S)$ as follows:*

1. $S \subseteq \mathfrak{F}(S)$
2. $\forall f \in \Sigma . f$ a constructor and $M_1,\ldots,M_k \in \mathfrak{F}(S) : f(M_1,\ldots,M_k) \in \mathfrak{F}(S)$
3. $\forall \delta \in \Sigma . \delta$ a destructor and $M_1,\ldots,M_k \in \mathfrak{F}(S) \wedge \delta(M_1,\ldots,M_k) \rightarrow M' \Rightarrow M' \in \mathfrak{F}(S)$

So by $\mathfrak{F}(S)$ we refer to the (infinite) set obtained from $S$ by applying functions symbols and performing reductions. This denotes all the "information" that can be generated from $S$.

### 3.1.3 Operational Semantics

We now introduce the semantics of our process calculus. We begin with two binary relations, $\alpha$-congruence and structural congruence. $\alpha$-congruence $\equiv_\alpha$ equates processes up to the safe renaming of bound names. Structural congruence $\equiv$ equates processes with identical structure, using both term equality $=_E$ and the free names of a process $fn(P)$ - names that aren't bound to inputs or restrictions. Both relations are auxiliary to the definition of the semantics of processes, even though structural congruence also plays a role in the semantics of the logic, as will be seen in Section 3.2.

We will then present the semantics of the calculus in two distinct formulations. We'll define reduction semantics, that express how a process can evolve on its own, and we'll define labelled transition semantics, that allows us to express how a process can evolve by interacting with its environment. Note that we defer the semantics of the attacker output to Section 3.1.4.

**Definition 3.1.9.** ***($\alpha$-congruence).*** *Given a process P and a name $p$ (or set of names $\bar{p}$):*

$$\begin{array}{lcl} (\nu n)P & \equiv_\alpha & (\nu p)P\{n \leftarrow p\} \\ m(x).P & \equiv_\alpha & m(p).P\{x \leftarrow p\} \\ let\, n = M \text{ } in\, P & \equiv_\alpha & let\, p = M \text{ } in\, P\{n \leftarrow p\} \end{array}$$

*Where $p$ (or $\bar{p}$) is such that it doesn't occur in $P$.*

**Definition 3.1.10.** ***(Structural congruence).*** *Structural congruence $\equiv$ is the least congruence relation on processes $(P,Q)$ such that:*

$$\begin{array}{l} P \equiv_\alpha Q \Rightarrow P \equiv Q \\ P|\mathbf{0} \equiv P \\ P|Q \equiv Q|P \\ P|(Q|R) \equiv (P|Q)|R \\ P+Q \equiv Q+P \\ P+(Q+R) \equiv (P+Q)+R \\ [M_1 = M_2].P \equiv [M_2 = M_1].P \\ n \notin fn(P) \Rightarrow P|(\nu n)Q \equiv (\nu n)(P|Q) \\ (\nu n)\mathbf{0} \equiv \mathbf{0} \\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \\ M =_E M' \Rightarrow let\, n = M \text{ } in\, P \equiv let\, n = M' \text{ } in\, P \\ M =_E M' \Rightarrow m\langle M\rangle.P \equiv m\langle M'\rangle.P \\ M_1 =_E M_1' \Rightarrow [M_1 = M_2].P \equiv [M_1' = M_2].P \end{array}$$

Our rules for structural congruence are standard in process calculi, with the exception of the rules that specifically refer to terms. In these cases, similarly to [2], we consider processes to be structurally equivalent when both manipulate identical terms. This is sensible because there shouldn't be a structural distinction between a process that outputs a value $x$ and a process that outputs dec(enc($x,k$)$,k$).

We can now define the reduction and labelled transition semantics of processes. The reduction semantics specify under which circumstances a process can perform a computational step, defining how a process can evolve within itself, in the sense that no interaction with the external environment is represented.

**Definition 3.1.11.** ***(Reduction semantics).*** *The reduction relation $P \longrightarrow Q$ over processes is defined as the least relation closed under the following rules:*

$$\frac{M \textit{ is destructor-free}}{\textit{let } n = M \textit{ in } P \longrightarrow P\{n \leftarrow M\}}$$

$$\frac{y \textit{ is destructor-free}}{x\langle y\rangle.P + R \mid x(z).Q + S \longrightarrow P|Q\{z \leftarrow y\}}$$

$$\frac{M_1 \textit{ and } M_2 \textit{ are destructor-free and } M_1 =_E M_2}{[M_1 = M_2].P \longrightarrow P}$$

$$\frac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$$

$$\frac{P \longrightarrow Q}{(\nu n)P \longrightarrow (\nu n)Q}$$

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

Our reduction rules are fairly standard, with the exception of the destructor-freedom conditions of the rule for the *let*, communication and testing constructs.

The “destructor-free” restrictions are a way of eliminating the ability of processes to communicate and handle terms that do not represent concrete values. The idea stems from the fact that destructors model computational steps that are bounded by some inherent conditions (defined by the rewrite rules). For instance, a symmetric decryption destructor can only be correctly applied to a value if the key used to decrypt is the same as the one used to encrypt. However, nothing forbids a process from trying to perform such an erroneous decryption. In these scenarios, we have a normal form in the sense that the term cannot be further reduced (since the destructor equations can’t be fired), but a normal form that does not represent a valid computation.

To handle these erroneous terms, instead of allowing processes to compute and communicate these error values amongst each other, we simply halt the progress of processes that compute or attempt to communicate such terms. Therefore, computation through the *let* rule can only proceed if the declared term can be written through $=_E$ as a destructor-free term. In the rule for communication, the same reasoning is applied. Communication can only proceed if the communicated term can be written as a destructor-free term. Finally, in the rule for testing, processes can only proceed if both terms to be tested can be written destructor-free. Such halting on invalid terms can be compared to a program raising an (uncaught) exception when it detects some computational error. These simple restrictions allow us to ensure that all values received by a process are valid and don't represent erroneous computations.

Reduction semantics are sufficient to completely define the operational behaviour of processes, but by construction lack the ability to consider how a process can potentially interact with its environment. Hence, we also present the more general labelled transition semantics that precisely define how a process can evolve by communicating with its environment, and subsume the reduction semantics.

The present labelled transition semantics is not intended to characterise a complete notion of behavioural equivalence (as could be expected), but rather to allow the observation of actions in our logic. Despite not belonging to the scope of this work, we can point out that our labelled semantics do not allow for a complete characterisation of equivalent behaviour, in the sense that our extrusion rules reveal information in such a way that they result in a higher discriminative power then that of behavioural equivalence. We leave further research on this topic to future work.

In this labelled transition relation, a transition label $\alpha$ can be a $\tau$, representing the silent action, an input $n(x)$, an output $n\langle M\rangle$ or a bound output $\nu\bar{x}.n\langle M\rangle$, where $\bar{x} \subseteq names(M)$.

**Definition 3.1.12.** ***(Labelled transition semantics).*** *The labelled transition relation $P \xrightarrow{\alpha} Q$ is defined as the least relation closed under the following rules:*

$$\frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q}\ \textit{(Tau)}$$

$$\frac{M \textit{ is destructor-free}}{n\langle M\rangle.P \xrightarrow{n\langle M\rangle} P}\ \textit{(Out)}$$

$$\frac{M \textit{ is destructor-free}}{n(x).P \xrightarrow{n(M)} P}\ \textit{(Inp)}$$

$$\frac{P \xrightarrow{\alpha} Q \qquad \forall n \in \bar{u}\text{: } n \notin names(\alpha)}{(\nu\bar{u})P \xrightarrow{\alpha} (\nu\bar{u})Q}\ \textit{(Res)}$$

$$\frac{P \xrightarrow{n\langle M\rangle} P' \qquad \bar{s} \subseteq names(M) \textit{ and } \bar{s} \subseteq \bar{u} \qquad \bar{u'} = \bar{u} \setminus \bar{s}}{(\nu\bar{u})P \xrightarrow{\nu\bar{s}.n\langle M\rangle} (\nu\bar{u'})P'}\ \textit{(BoundOut)}$$

$$\frac{P \equiv P' \qquad P' \xrightarrow{\alpha} Q' \qquad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \textit{(Cong)}$$

Again, our labelled semantics are defined with standard rules, with the added destructor-freedom constraints as in the reduction semantics. The subsumption of reduction is embodied in the Tau rule, where all reductions are labelled with the $\tau$ label. The Out rule allows the output of valid terms to the environment and the Inp rule allows the input of equally valid terms. The more complex rules handle restriction: The Res rule allows outputs to proceed if the term doesn't mention the restricted names; the BoundOut rule handles output of terms that contain restricted names by extruding the scope of the necessary restrictions.

### 3.1.4 Defining Attackers

Using the calculus of the previous sections, we can easily encode passive attackers - an attacker that only eavesdrops on messages and attempts to perform operations on them. The definition of the attacker "code" itself must be done on a per-protocol basis, but a general schema can be defined. These simple attacker models are sufficient, since we can then use our logic to reason locally about the information the attacker can obtain.

**Definition 3.1.13.** ***(Passive Attacker Schema).*** *Given a protocol that exchanges $n$ messages over channel $c$, we can define a passive attacker PAttacker for this protocol as:*

$$\mathit{PAttacker} \triangleq (\nu\ \mathit{store})c(x_1).c\langle x_1\rangle.\ldots.c(x_n).c\langle x_n\rangle.\mathit{store}\langle x_1,\ldots,x_n\rangle$$

Attackers defined in such a way collect the several messages that are exchanged throughout a run of the protocol (forwarding them back so the protocol can proceed) and store the messages, here modeled by the output on channel store. In practice we don't need to output on a restricted name, just on a name that isn't used by the protocol.

While such a process does not perform any operations with the messages themselves, one could encode such operations in the process itself, or as will be seen in Section 3.2, we can reason about these operations implicitly.

The calculus can also be extended to model active attackers. This type of attackers not only eavesdrop on communication that takes place within a protocol but can also inject messages at any point of the protocol run. These messages can consist of any value obtained by the attacker, either through past inputs, or by performing computation over past inputs. Since we cannot statically determine values that will be obtained dynamically by the attacker, we extended the calculus' syntax with a new action prefix that models such an output.

**Definition 3.1.14.** ***(Extended Calculus Syntax).***

$$\begin{array}{lcll} P, Q & ::= & \ldots & \\ & & & \\ \alpha & ::= & \ldots & \\ & | & m\langle * \rangle & \textit{(Attacker Output)} \\ & & & \\ M & ::= & \ldots & \end{array}$$

We define the semantics of such an operation through a new structural congruence rule. The rule allows us to transform the Attacker Output action into a choice, where each choice branch contains an output of a message that the process can compute.

$$t_i \in \mathfrak{F}(S) \Rightarrow m\langle * \rangle.P \equiv m\langle t_1 \rangle.P + \cdots + m\langle t_n \rangle.P$$

Where $S$ is the set of terms that occur in $P$ that are values. We therefore have a process that can non-deterministically output any value it can compute. An obvious "practical" problem with this approach is that the cardinality of the set $\mathfrak{F}(S)$ is infinite, therefore an implementation of such an operation must truncate the number of possible choice branches (as will be seen in Chapter 4).

With this new operation we can extend our passive attacker schema to implement active attackers. The idea is that instead of simply forwarding the message back to the protocol, it performs the new attacker output action, therefore outputting any value it can construct up to that point.

**Definition 3.1.15.** ***(Active Attacker Schema).*** *Given a protocol that exchanges n messages over channel c, we can define an active attacker AAttacker for this protocol as:*

$$AAttacker \triangleq (\nu\ store)c(x_1).c\langle * \rangle.\ldots.c(x_n).c\langle * \rangle.store\langle x_1, \ldots, x_n \rangle$$

### 3.1.5 Example of a Protocol

We present in Figure 3.2 an example of a simple protocol written in the calculus. Considering an equational theory consisting of the equation for symmetric cryptography, the protocol consists of two principals, *Alice* and *Bob*, that share a symmetric key and communicate over a public channel $c$.

In this toy protocol, the principal Bob generates a session key using the name restriction operator. Name restriction can be used as name generation, since when we restrict a name to a certain process we're stating that such a name is guaranteed to be fresh, in the sense that no other process can capture it (until the generating process communicates the name). After generating the key, Bob encrypts it using the key it shares with Alice, and sends it over channel c. Alice, who is waiting for an input on that channel, reads the cypher-text and then performs

$$\text{Alice}(k) \triangleq c(m).\text{let } n = \text{dec}(m,k) \text{ in } c\langle \text{enc}(\text{hello}, n)\rangle.\mathbf{0}$$

$$\text{Bob}(k) \triangleq (\nu\ \text{key})c\langle \text{enc}(\text{key}, k)\rangle.c?(x).\left[\text{dec}(x,\text{key}) = \text{hello}\right].\text{ok}\langle\rangle.\mathbf{0}$$

$$\text{Attacker} \triangleq c(x).c\langle *\rangle.c(y).c\langle *\rangle.\mathit{store}\langle x,y\rangle.\mathbf{0}$$

$$\text{System} \triangleq (\nu\ \text{sharedKey})(\text{Alice}(\text{sharedKey}) \mid \text{Bob}(\text{sharedKey}))$$

$$\text{World} \triangleq \text{System} \mid \text{Attacker}$$

Figure 3.2: A Simple Protocol

a decryption operation on the cypher-text using the shared key. Assuming the decryption succeeds, Alice proceeds to send, over channel c, the value *hello* cyphered with the fresh session key and terminates. Bob will then receive this cypher-text and test if the result of decrypting the message with the shared key results in the value *hello*, if it does, it will signal such a fact by outputting over channel *ok* and terminating. We can imagine the communication of the value *hello* to model the communications that Alice and Bob wanted to assure as secure.

This simple example illustrates the general ideas of how we can use the calculus to model the standard techniques used in security protocols, such as generating session keys (the same mechanics apply to nonce generation), pre-determined key sharing, encrypted communication and performing tests on received data. We also present an active attacker on the protocol, following the schema of Section 3.1.4. While in this brief example we simply decrypt a value and test it, we could as easily test the digital signature on a message using an asymmetric cryptography scheme.

## 3.2 Logic

Having established our process calculus model of Section 3.1, we can now motivate and introduce our logic. Our goal was to develop a logic with which to reason about security protocols and their correctness both in the absence and presence of attackers. We therefore needed to have enough generality to state these correctness properties while considering just the protocol itself and the protocol combined with attackers.

The idea was to explore some of the key aspects of dynamic spatial logics, such as local and compositional reasoning, in order to achieve a logic that could reason locally about the several principals of the protocol as well as reason about the protocol in the presence of attackers. Despite this, as motivated in Section 2.2.6.4, these logics were not designed to consider a model with structured terms like ours. In such a model, properties that strictly refer to names, be it name communication or sharing, can become vacuous since they do not consider the vulnerabilities nor the security benefits that the use of cryptography supplies. Also, since we can consider several equational theories, the denotation of communicating a term that contains a name can vary.

However, we do want the ability to reason separately about a protocol and its attacker(s). We may want to verify that the message exchanges in the protocol are valid, without the presence of attackers, or we may want to verify that the same happens in the presence of attackers, in the sense that an attacker cannot harmfully tamper with the protocol in such a way that it will obtain privileged information (like keys), or be able to impersonate a principal.

To achieve this, we propose an extension to a core spatial behavioural logic that enables reasoning at the term level. The idea is that the information a process can know depends on what terms it has obtained, or that it can generate. For instance, a process that knows a cypher-text and doesn't know the adequate cypher-key can never apply a decryption operation on the cypher-text to obtain the clear-text, while a process that knows both a cypher-text and the cypher-key can always obtain (or know) the clear-text.

Therefore, we combine the power of spatial-behavioural logics with the ability to reason about the information (terms) a process can derive by applying functions and reductions to terms it already has, coupling the ability to reason about properties of spatial arrangement and behaviour to the ability to reason about the derivable information modulo the considered equational theories. Hence allowing us to reason about the spatial and behavioural dynamics of agents and about the information they can obtain and compute.

Our extension of a spatial behavioural logic (presented in [11]) consists of two logical connectives that deal (epistemically) with terms. The *knowledge* predicate expresses the fact that a process either holds the given term in its "code" or it can derive such a term from those it does hold (by combining the terms through functions). The *secret* quantifier allows the quantification over terms that are secret in a process, that is, terms that are unforgeable by other processes.

### 3.2.1 Syntax

We now introduce the syntax of our logic, followed by its semantics, some derived idioms and examples of potentially interesting properties.

**Definition 3.2.1.** ***(A spatial-behavioral logic with a knowledge modality).*** *Given $(A, B)$ formulas, $(x, y, z)$ name variables, $X$ propositional variable ($\eta$ is a meta-variable that denotes a name or a name variable), $\alpha$ (action), $t$ (term) and $(\varphi, \psi)$ knowledge formulas we define the syntax of our logic as presented in Fig. 3.3.*

Since we only allow conjunction of knowledge formulas, a knowledge formula can be read as a denotation of a set of terms, as shown in Definition 3.2.2.

$$
\begin{array}{rcll}
A, B & ::= & \mathbf{T} & \text{(True)} \\
& \mid & \eta = \eta' & \text{(Equality)} \\
& \mid & \neg A & \text{(Negation)} \\
& \mid & A \wedge B & \text{(Conjunction)} \\
& \mid & \mathbf{0} & \text{(Void)} \\
& \mid & A|B & \text{(Composition)} \\
& \mid & nA & \text{(Revelation)} \\
& \mid & \reflectbox{N}x.A & \text{(Fresh quantification)} \\
& \mid & \forall x.A & \text{(Name quantification)} \\
& \mid & \alpha.A & \text{(Action)} \\
& \mid & X & \text{(Propositional variable)} \\
& \mid & \nu X.A & \text{(Greatest fixpoint)} \\
& \mid & \mu X.A & \text{(Least fixpoint)} \\
& \mid & \mathbb{K}\varphi & \text{(Knowledge)} \\
& \mid & \mathsf{S}x.A & \text{(Secret quantification)} \\
\\
\varphi, \psi & ::= & \varphi \wedge \psi & \text{(Conjunction)} \\
& \mid & t & \text{(Term)} \\
& \mid & \top & \text{(True)}
\end{array}
$$

Figure 3.3: Logic Syntax

**Definition 3.2.2.** ***(Denotation of Knowledge Formulas).*** *We define the denotation of a knowledge formula $\varphi$, written as $[\varphi]$, inductively on the structure of the formula:*

$$\begin{array}{lcl} [\top] & \triangleq & \emptyset \\ [\varphi \wedge \psi] & \triangleq & [\varphi] \cup [\psi] \\ [t] & \triangleq & \{t\} \end{array}$$

Informally, our new connectives (knowledge and secret quantification) are interpreted as follows: a process satisfies $\mathbb{K}\varphi$ if it can obtain the terms denoted by $\varphi$ from the terms it possesses; a process satisfies $\mathsf{S}x.A$ if it possesses a term $t$ that contains a restricted name (a term that cannot be forged by other processes), where by instantiating the quantified variable $x$ with the term $t$, the process satisfies formula $A$.

### 3.2.2 Semantics

We will now move towards the semantics of the logic, for which some intermediary definitions are necessary. Following the presentation of the spatial logics of [14, 11], the semantics are given in a domain of property sets (Psets). A Pset (Def. 3.2.3) is a set of processes closed under structural congruence with *finite support* that represents the set of processes in which the formula holds. The support of a Pset is a finite set of names such that the Pset is closed by transposition of names out of the support. The idea is to give a uniform treatment to fresh names. The support also provides a bound on the set of free names of a formula. We denote by $\mathbb{P}$ the collection of all Psets.

The denotation of an arbitrary formula $A$ is given by a Pset $[[A]]_v$, where $v$ is a valuation. A valuation assigns to each propositional variable free in $A$ a Pset. We write $fn^v(A)$ as the set of free names of $A$ under $v$. Informally, $fn^v(A)$ contains the free-names of formula $A$, adding the (least) support of the valuation mapping for each free propositional variable occuring in $A$. This definition is used in the semantics of the fresh name quantifier, which requires the witness to be fresh for the Pset denoted by a formula.

**Definition 3.2.3.** ***(Property set).*** *[14] A property set is a set of processes $\Psi$ such that:*

- *For all $Q$, if $P \in \Psi$ and $P \equiv Q$ then $Q \in \Psi$*
- *There is a finite set of names $N$ such that for all $n, m \notin N$, if $P \in \Psi$ then $P\{n \leftrightarrow m\} \in \Psi$.*

*For a detailed account of Psets, we refer to [14].*

To encode the proper denotation for $\mathbb{K}\varphi$, some auxiliary definitions are required. Informally, the idea is that we want to reason about the terms that exist throughout the process' code. We must therefore abstract away the details of the underlying model and focus specifically on terms. To achieve this, we define a relation $\vdash_k$ on processes and knowledge formulas that given a process extracts the set of terms present in the process, encoding it in a knowledge formulas.

The resulting knowledge formula can be viewed as a set of terms known by the process, not considering the equational theory.

When defining the $\vdash_k$ relation, some care is needed due to the dual meaning of a term. When a term is built from constructors, such as *enc(x,y)*, its not necessary that a process that holds such a term knows *x* and *y*, since *enc(x,y)* might have been received from some other process. However, when dealing with terms with destructors, which model function application *per se*, the several subterms must be known so the application can take place. For instance, a term such as *dec(enc(x,y),y)* requires that both subterms (*enc(x,y)* and *y*) are also known. Generally all terms are normal forms either built from constructors or reduced by one (or more) destructors. However, when dealing with destructors that have *name variables*, such as in the process:

$$a(x).a\langle \mathrm{dec}(x,k)\rangle.\mathbf{0}$$

where *x* is a variable bound to the input, the term *dec(x,k)* is not known *per se*, since it contains a name that is still “unresolved”. In fact, after variable *x* is instantiated by the input term, dec(x,k) might not be a normal form anymore (if the received term is some piece of data encrypted with key *k*). To this extent, it would be unnatural for the example process to know the term dec(x,k), since *x* is a name local to the process. Therefore when we define $\vdash_k$, we use an auxiliary operation on knowledge formulas to remove terms with bound names. However, this removal cannot be completely blind. Going back to the example process, while the term *dec(x,k)* is still “incomplete” and should be removed from $\vdash_k$, it does indeed represent an application of a destructor, which in itself provides information about the knowledge of the process that cannot be overlooked.

A similar scenario occurs when considering an equational theory with pairs, as given by the equations:

$$\mathrm{first}(\mathrm{pair}(x,y)) = x$$

$$\mathrm{snd}(\mathrm{pair}(x,y)) = y$$

And the process:

$$a(x).a\langle \mathrm{pair}(dec(x,k),\mathrm{info})\rangle.\mathbf{0}$$

In the term output, seeing as the pairing depends on the to-be input name, it must be removed. However, *k* must somehow remain in the extracted terms, since its being used in a destructor inside the pairing.

To account for these situations in $\vdash_k$, we use the procedure *sub*, while the sub-formula removal is handled by the $\uparrow$ operation.

**Definition 3.2.4.** ***(Relevant subterm extraction).*** *We define the relevant subterm extraction operation on a term M - $sub(M)$ - as follows ($f$ denotes constructors and $\delta$ destructors):*

$$\frac{}{sub(\delta(t_1,\ldots,t_n)) \triangleq sub(t_1)\cup\cdots\cup sub(t_n)} \qquad \frac{n \text{ is not input-bound}}{sub(n) \triangleq n}$$

$$\frac{\nexists t_i \text{ with an input-bound name or a destructor}}{sub(f(t_1,\ldots,t_n)) \triangleq f(t_1,\ldots,t_n)} \qquad \frac{n \text{ is input-bound}}{sub(n) \triangleq \emptyset}$$

$$\frac{\exists t_i \text{ with an input-bound name or a destructor}}{sub(f(t_1,\ldots,t_n)) \triangleq sub(t_1) \cup \cdots \cup sub(t_n)}$$

Our relevant subterm extraction procedure is sound, in the sense that it never produces knowledge formulas containing destructor symbols (terms that aren't actual values), as stated in Lemma 3.2.1.

**Lemma 3.2.1.** ***(Destructor-Freedom of Subterm Extractions).*** *For all terms $t$, $sub(t) = \varphi$ is such that $\varphi$ does not contain terms with destructor function symbols.* Proof: *Appendix A.1*

**Definition 3.2.5.** ***(Name occurrence sub-formula removal).*** *We define the removal of sub-formulas of a formula $\varphi$, in which the name $x$ occurs, $\varphi \uparrow x$, inductively on the structure of formulas:*

$$\begin{aligned}
(\varphi \wedge \psi) \uparrow x &\triangleq (\varphi \uparrow x) \wedge (\psi \uparrow x) \\
t \uparrow x &\triangleq t \text{ if } x \notin names(t) \\
t \uparrow x &\triangleq \top \text{ if } x \in names(t) \\
\top \uparrow x &\triangleq \top
\end{aligned}$$

With the issue of relevant subterms handled, we can now fully define the term extraction operation on processes.

**Definition 3.2.6.** ***(Term Extraction).*** *Given a process $P$, we construct a knowledge formula $\varphi$ denoting the set of all terms that the process holds in its "code" with the relation $P \vdash_k \varphi$, closed by the following rules on the structure of processes:*

$$\frac{P \vdash_k \varphi \quad Q \vdash_k \psi}{P + Q \vdash_k (\varphi \wedge \psi)}$$

$$\frac{P \vdash_k \varphi \quad Q \vdash_k \psi}{P|Q \vdash_k (\varphi \wedge \psi)}$$

$$\frac{P \vdash_k \varphi}{n(x).P \vdash_k \varphi \uparrow x}$$

$$\frac{P \vdash_k \varphi}{x\langle M\rangle.P \vdash_k \varphi \wedge sub(M)}$$

$$\frac{P \vdash_k \varphi}{(\nu n)P \vdash_k (\varphi \uparrow n)}$$

$$\frac{P\{n \leftarrow M\} \vdash_k \varphi}{let\, n = M \text{ in } P \vdash_k \varphi \wedge sub(M)}$$

$$\frac{P \vdash_k \varphi}{[M = N].P \vdash_k \varphi \wedge sub(M) \wedge sub(N)}$$

$$\frac{P \vdash_k \varphi}{(\mathbf{rec}\, X.P) \vdash_k \varphi}$$

$$\frac{}{X \vdash_k \top}$$

$$\frac{}{\mathbf{0} \vdash_k \top}$$

The extraction procedure itself simply collects the terms (and their relevant subterms) as they occur in the process, applying the removal operation when a bound name is found. Our extraction procedure aims at collecting only values from the process, as validated by Theorem 3.2.1.

**Theorem 3.2.1.** ***(Characterization of Term Extraction).*** *Given a process $P$ such that $P \vdash_k \varphi$, the knowledge formula $\varphi$ is made up of terms that are values, that is, composed solely of names and/or constructor function symbols.* Proof: *Appendix A.2*

We now need to define a notion of deducibility for knowledge formulas, where one formula can be deduced from another if the terms denoted by the deduced formula are contained in the functional equational closure of the set of terms denoted by the original formula (Def. 3.2.7).

**Definition 3.2.7.** ***(Deducibility of knowledge formulas).*** *Given knowledge formulas $\psi$ and $\varphi$, we define the deduction relation of knowledge formulas, written $\psi \models \varphi$ as:*

$$\begin{array}{rcl} \psi \models \top & \triangleq & \top \\ \psi \models \varphi_1 \wedge \varphi_2 & \triangleq & \psi \models \varphi_1 \wedge \psi \models \varphi_2 \\ \psi \models t & \triangleq & t \in \mathfrak{F}([\psi]) \end{array}$$

With all the necessary preliminary definitions completed, the semantics of the logic can be introduced.

**Definition 3.2.8.** ***(Denotation of formulas).*** *The denotation of a formula $A$ is defined by the Pset $[\![A]\!]_v$ as follows:*

$$\begin{array}{rcl}
[\![T]\!]_v & \triangleq & \{P \mid P \in \mathbb{P}\} \\
[\![n = m]\!]_v & \triangleq & \mathit{if}\, n = m\ \mathit{then}\ \mathbb{P}\ \mathit{else}\ \emptyset \\
[\![\neg A]\!]_v & \triangleq & \mathbb{P} \setminus [\![A]\!]_v \\
[\![A \wedge B]\!]_v & \triangleq & [\![A]\!]_v \cap [\![B]\!]_v \\
[\![A|B]\!]_v & \triangleq & \{P \mid \exists Q, R.P \equiv Q|R\ \mathit{and}\ Q \in [\![A]\!]_v\ \mathit{and}\ R \in [\![B]\!]_v\} \\
[\![n\circledR A]\!]_v & \triangleq & \{P \mid \exists Q.P \equiv (\nu n)Q\ \mathit{and}\ Q \in [\![A]\!]_v\} \\
[\![\forall x.A]\!]_v & \triangleq & \cap_{n\in\Lambda} [\![A\{x \leftarrow n\}]\!]_v \\
[\![\text{И} x.A]\!]_v & \triangleq & \cup_{n\notin fn^v(A)}([\![A\{x \leftarrow n\}]\!]_v \setminus \{P \mid n \in fn(P)\}) \\
[\![\alpha.A]\!]_v & \triangleq & \{P \mid \exists Q.P \xrightarrow{\alpha} Q\ \mathit{and}\ Q \in [\![A]\!]_v\} \\
[\![X]\!]_v & \triangleq & v(X) \\
[\![\nu X.A]\!]_v & \triangleq & \cup\{\Psi \in \mathbb{P} \mid \Psi \subseteq [\![A]\!]_{v[X\leftarrow\Psi]}\} \\
[\![\mu X.A]\!]_v & \triangleq & \cap\{\Psi \in \mathbb{P} \mid [\![A]\!]_{v[X\leftarrow\Psi]} \subseteq \Psi\} \\
[\![\mathbb{K}\varphi]\!]_v & \triangleq & \{P \mid P \vdash_k \psi\ \mathit{and}\ \psi \models \varphi\} \\
[\![\mathsf{S}x.A]\!]_v & \triangleq & \{P \mid \exists Q, t.P \equiv (\nu k)Q\ \mathit{and}\ Q \in [\![A\{x \leftarrow t\}]\!]_v \\
 & & \mathit{and}\ Q \vdash_k \phi\ \mathit{such\ that}\ t \in [\phi]\ \mathit{and}\ k \in \mathit{names}(t)\}
\end{array}$$

The denotations for the behaviour and spatial operators of the logic are the same as presented in [11]. The boolean connectives, name equality, least and greatest fix-points and first-order universal quantification are defined standardly. The composition, or separation operator $A \mid B$, allows us to state that a process can be structurally decomposed in two processes, where one satisfies formula $A$ and the other formula $B$. This allows reasoning about the subsystems of larger systems - *local* and *compositional* reasoning. The revelation operator $n\circledR A$, states that a process has a restricted name $n$ and the process under the restriction satisfies formula $A$. This operator allows us to reason under restricted names and (combined with the fresh name quantifier) allows us to reason about these hidden names as well. The action prefix formula $\alpha.A$ holds in a process that can perform an action with label $\alpha$ and whose continuation satisfies $A$. For the knowledge predicate, we define that such a formula holds true in a process from which its knowledge-formulaic-encoding of terms (given by $\vdash_k$) can be used to deduce the given knowledge formula, that is, if we take the terms the process holds, we can combine them through function application and term reduction in such a way that we obtain the terms denoted by the given knowledge formula. This technique implicitly models the ability to perform computation on data to obtain new data (such as applying decryption to received values). For the secret quantifier, we require that the process has some restricted name $k$, such that the witness is a term occurring in the process that lies under the restriction and contains $k$ as a subterm. This way we are sure that the witness is indeed a term that is secret of the process, in the sense that only that process can have generated it. By convention, in the action prefix formula, if $\alpha$ is empty (written as $\Diamond$) then all possible actions are considered. Such a formula is usually called *next*. Also, an alternative syntax (from HML [34]) for the action prefix formula can be used: $\langle\alpha\rangle A$.

The satisfaction relation, given the denotation of a formula, is defined standardly as follows.

**Definition 3.2.9.** ***(Satisfaction of formulas).*** *Given the set of processes $[\![A]\!]_v$, a process $P$ satisfies formula $A$ under valuation $v$ whenever $P \in [\![A]\!]_v$. Written $P \models_v A$.*

Given the already introduced logic, its useful to derive some standard (from spatial and behavioural logics) idioms such as a quantifier over hidden names $\mathsf{H}$, a free name occurrence predicate $@$, existential quantification of names $\exists$, an "all next" modality $\boxdot$, a "always in the future" modality $\Box$ and a "eventually in the future" modality $\Diamond$.

**Definition 3.2.10.** ***(Derived formulas).***

$$\begin{array}{llll}
\mathsf{H}x.A & \triangleq & \reflectbox{N}.x \circledR A & \textit{(Hidden name quantification)} \\
@\eta & \triangleq & \neg\eta \circledR T & \textit{(Free name predicate)} \\
\exists x.A & \triangleq & \neg\forall x.\neg A & \textit{(Existential name quantification)} \\
\boxdot A & \triangleq & \neg\Diamond\neg A & \textit{(All next modality)} \\
\Box A & \triangleq & \nu X.(A \wedge \boxdot X) & \textit{(Always modality)} \\
\Diamond A & \triangleq & \mu X.(A \vee \Diamond X) & \textit{(Eventually modality)}
\end{array}$$

We also define some commonly used formulas. A formula Public $\triangleq \mathsf{H}x.\neg @x$, that states that a process has no "real" restricted name, in the sense that no hidden names occur in the process; using recursion, we can define a formula that states that the subsystem obtained after removing all restricted names satisfies some property $A$:

$$\text{inside}(A) \triangleq \nu X.((\text{Public} \wedge A) \vee \mathsf{H}x.(@x \wedge X))$$

The inside property is useful when we want to look under all name restrictions without quantifying over them explicitly.

## 3.2.3 Example

We will now recall on the example of Section 3.1.5, Figure 3.2, and state some properties about that system using our newly defined logic. When considering the system without the presence of attackers, we still want to make sure that the message exchanges are such that both Alice and Bob agree on the session key and exchange the *hello* message, which can be verified by checking that it is always true that eventually the system will be able to perform the output on channel *ok*. This property, which we can call *correctness* can be stated as:

$$\text{correctness} \triangleq \Box\Diamond\langle ok\rangle\ \text{True}$$

Since all processes satisfy *True*, the *correctness* property has the intended meaning. Seeing as the example is very simple, we can trivially verify that System $\models$ correctness. However, we don't want to be limited to properties of systems without the presence of attackers. For this we will consider the Alice and Bob system composed with the Attacker process, in the process World.

We want to verify that only Alice and Bob can obtain the value *hello*, since we're considering that value as the privileged information that needs to be kept confidential. We will begin by defining a property *threeHellos*, that states that when we eventually look inside the system, by lifting the restricted names, we obtain three subsystems that know the value *hello*.

$$\text{threeHellos} \triangleq \Diamond\text{inside}(\mathbb{K}\,\text{hello} \mid \mathbb{K}\,\text{hello} \mid \mathbb{K}\,\text{hello})$$

What we now want to verify is that it is never the case that in our *World*, the property *threeHellos* holds, so:

$$\text{World} \models \Box\neg\text{threeHellos}$$

Such is the case since we know that both Alice and Bob will know *hello*, but since the keys are never obtained by the attacker, it can never obtain the value *hello*. We can also state that the attacker never obtains any of the keys used in the protocol. We first define a property *keyLeak*, that uses quantification over the hidden names (the keys) to state that we have a system where 2 subsystems know both keys, and another subsystem (the attacker) knows one or the other:

$$\text{keyLeak} \triangleq \text{ИshK}.\ \text{ИsK}.\ (\mathbb{K}(\text{shK} \wedge \text{sK}) \mid \mathbb{K}(\text{shK} \wedge \text{sK}) \mid (\mathbb{K}\text{shK} \vee \mathbb{K}\text{sK}))$$

To state that the keys are indeed never leaked, we need to check if:

$$\text{World} \models \Box\neg\text{keyLeak}$$

With these short examples we hope to transmit some of the ways through which we can state relevant properties about protocol correctness using our logic, with or without the presence of attackers.

## 3.3 A Syntactical Approach to Knowledge Deduction

For the denotation of $\mathbb{K}\varphi$, the functional and equational closure of the set of terms that occur in the process is used. This approach, while being purely declarative and hence more adequate for denotational semantics, has the inconvenience that such a closure will always be a set of infinite cardinality (assuming a non-empty signature), resulting in what at first glance may appear as undecidable semantics.

To this end, in this section we present a syntactical approach for entailment of knowledge formulas, defined by a proof system formulated with a sequent calculus of knowledge formulas. The calculus is equipped with rules generated from the equational theory in order to consider the ability to combine terms to generate new information, therefore being able to represent the full scope of derivable information. We then present soundness (Theorem 3.3.2) and completeness

(Theorem 3.3.6) results for our proof system. From our completeness proof we also obtain an algorithm (Definition 3.3.5) that computes a sound finite approximation of the functional and equational closure of a set of terms (Theorem 3.3.4), thus proving the decidability of our proof system and enabling the implementation of Chapter 4.

Each rule of our calculus therefore represents a possible computational step that a principal (or an attacker) can perform on a term in order to produce a new term. Our sequents obey to the following schema $\Gamma \vdash A$, where $\Gamma$ is a set of formulas and $A$ is an individual formula (similar to sequent calculi formulations of proof systems for intuitionistic logic). The meaning of a sequent is that if we can build a proof derivation for $\Gamma \vdash A$, then the set of formulas $\Gamma$ entail formula $A$.

**Definition 3.3.1.** ***(Proof System K for knowledge formulas).*** *With $\Gamma$ being a set of logical formulas, $A$, $B$ and $C$ individual logic formulas, $\Sigma$ a signature and $E$ an equational theory, the sequent calculus formulation for our proof system $K$ for knowledge formulas is defined by the following set of rules:*

$$\frac{}{\Gamma, A \vdash A}\,(Id)$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\,(\wedge\text{: left}) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge\text{: right})$$

*For every constructor function symbol $f$ with arity n, such that $f \in \Sigma$:*

$$\frac{\Gamma \vdash t_1 \ldots \Gamma \vdash t_n}{\Gamma \vdash f(t_1, \ldots, t_n)}\,(funRight) \qquad \frac{\Gamma, f(t_1, \ldots, t_n) \vdash C}{\Gamma, t_1, \ldots, t_n \vdash C}\,(AttLeft)$$

*For every equation $f(t_1, \ldots, t_n) = s \in E$:*

$$\frac{\Gamma, s \vdash C}{\Gamma, f(t_1, \ldots, t_n) \vdash C}\,(DestrLeft)$$

Rules (Id), ($\wedge$: left) and ($\wedge$: right) are standard derivation schemas for conjunction and identity, from intuitionistic logic. Rule (funRight) represents constructor application on the right-hand-side of a sequent, stating that we can always entail a constructor application from knowledge formulas $\Gamma$ if we can entail each of the function operands from the set of formulas. Rule (AttLeft) states that any formula we can prove from a set of formulas $\Gamma$ appended with a constructor application term can also be proved from $\Gamma$ appended with each of the operands of the function application. Rule (DestrLeft) expresses the equalities induced by the considered equational theory. The idea is that anything that can be proved by a term $f(t_1, \ldots, t_n)$ can also be proved by a term $s$ if there is an equation such that both terms are equal.

With such a deductive system, we wish to demonstrate through a purely syntactic manner when a knowledge formula is a logical consequence of another, in the sense that using the considered equational theory and function applications over the terms denoted by a formula we can produce the terms denoted by another. Formally, we want our deductive system K to be *sound* - whenever $A \vdash B$ then $A \models B$, and *complete* - whenever $A \models B$ then $A \vdash B$. We will now move towards the proof that our deductive system of knowledge formulas is indeed well behaved with regard to our semantics. We establish that the system is both sound and complete in this sense, by first proving some preliminary proof-theoretic lemmas that aim at assisting the main proofs. Additionally, we define an algorithmic procedure that computes a finite approximation to the functional equational closure of a set of terms and prove that the approximation is correct, in the sense that it can accurately represent all terms in the closure. We also prove the termination of the procedure, thus proving the decidability of our knowledge formula semantics.

Our first results relate to the admissibility of certain rules, such as *Weakening* and *Cut*. A rule is admissible in a deductive system if it does not add to the deductive power of the system, that is, adding the rule does not allow us to prove sequents that would be previously unprovable. Weakening is a rule that states that we can add arbitrary elements to a sequence of formulas, which intuitively holds since we can always add assumptions to a proof. The Cut rule is a standard rule in sequent calculi, equivalent to a variety of rules in other proof theories, that states that given a sequent $\Gamma \vdash C$ and a sequent $\Gamma, C \vdash A$ we can derive $\Gamma \vdash A$. The idea is that we can "cut", or eliminate the formula $C$ out of the proof derivation. While this rule makes perfect sense from an intuitive sense, the Cut rule allows the introduction previously unseen formulas (formula $C$) in a proof derivation, which from a proof search perspective can be problematic. Cut admissibility therefore states that the Cut rule is expendable since it does not add to the deductive power of the proof system.

We require Cut admissibility to allow transitivity in the sequent calculus, as used in the proof of Theorem 3.3.5. Nevertheless, such a result is also interesting in a theoretical sense because it implies that all proofs have a "normal form" in which "all concepts required for the proof would in some sense appear in the conclusion of the proof" [32].

**Lemma 3.3.1.** ***(Admissibility of Weakening in K).*** *If a sequent $\Gamma \vdash A$ is derivable in K then $\Gamma, B \vdash A$ is also derivable in K, for any formulas $A, B$ and formula set $\Gamma$.*

*Proof:* Induction on the structure of $\Gamma \vdash A$. The proof follows trivially from the induction hypothesis in each derivation of $\Gamma \vdash A$.

Lemma 3.3.1 is a standard proof-theoretic result in sequent calculi, stating that the Weakening rule does not add to the power of the system. While not being a fundamental lemma for our results, it is an useful result that simplifies future proofs, where it is convenient to be able to add formulas to a proof derivation.

We will now head towards the proof of the admissibility of Cut. The idea for the proof is similar to that of [13]. We begin by defining a proof system KC (Definition 3.3.2) which consists of our original proof system K extended with the Cut rule, and we prove (Lemma 3.3.3) that for all derivations in KC that contain a single instance of the Cut rule, we can build the same derivation cut-free. Since a cut-free derivation in KC is a derivation in K, we conclude with Theorem 3.3.1, which establishes the admissibility of the cut rule in our original proof system K.

**Definition 3.3.2.** ***(System KC).*** *Let KC be the proof system obtained by adding to the rules of system K the following rule:*

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \textit{(Cut)}$$

*Where $A$ is called the cut formula.*

**Lemma 3.3.2.** ***(Admissibility of Weakening in KC).*** *If a sequent $\Gamma \vdash A$ is derivable in KC then $\Gamma, B \vdash A$ is also derivable in KC, for any formulas $A, B$ and formula set $\Gamma$.*

*Proof:* Induction on the structure of $\Gamma \vdash A$. The proof follows trivially from the induction hypothesis in each possible derivation of $\Gamma \vdash A$.

**Definition 3.3.3.** ***(Single-cut derivation).*** *A derivation $\Pi$ in $KC$ is called a single-cut derivation if it contains a single instance of the (Cut) rule, at the root of the derivation.*

**Lemma 3.3.3.** ***(Cut Lemma).*** *If a sequent $\Gamma \vdash A$ has a single cut derivation in $KC$ then it has a cut-free derivation in KC.*
*Proof: Appendix A.3.*

**Theorem 3.3.1.** ***(Cut Admissibility in System K).*** *If sequents $\Gamma \vdash A$ and $\Gamma, A \vdash C$ are derivable in K, then $\Gamma \vdash C$ is also derivable in K, for any formulas $A, C$ and formula set $\Gamma$.*

*Proof:* Since we're assuming that sequents $\Gamma \vdash A$ and $\Gamma, A \vdash C$ are derivable in K, we can safely assume them to be derivable in KC, since the derivation rules of $K$ are a subset of the derivation rules of $KC$. We can also safely assume that there exist cut-free derivations of $\Gamma \vdash A$ and $\Gamma, A \vdash C$ in $KC$, since such sequents can be derived in $K$ (that doesn't have (Cut)).

In $KC$, we can then apply the (Cut) rule to the cut-free derivations of $\Gamma \vdash A$ and $\Gamma, A \vdash C$ in order to produce a single-cut derivation of the sequent $\Gamma \vdash C$. By Lemma 3.3.3, if sequent $\Gamma \vdash C$ has a single-cut derivation in $KC$, then it has a cut-free derivation $\Pi$ in $KC$. Therefore, since $\Pi$ is cut-free, it also is a derivation in $K$, seeing as $K$ is obtained from $KC$ by removing the (Cut) rule, and thus we prove our the existence of a derivation in $K$ for $\Gamma \vdash C$. □

Given these preliminary results, we can now build towards the aforementioned soundness, completeness and decidability results for our proof system. We begin with the soundness of our deductive system.

**Theorem 3.3.2.** ***(Soundness of K).*** *If $\Gamma \vdash A$ is derivable in the proof system K, then $\Gamma \models A$.*
*Proof: Appendix A.4.*

Through Theorem 3.3.2, we establish that all proof derivations obtainable through our sequent calculus are sound w.r.t our intended term semantics. That is, each step in a derivation in our sequent calculus models an operation (computation) over terms that produces a new term in such a way that the resulting term is also in the functional equational closure.

We will now proceed towards an algorithm that computes a finite approximation of $\mathfrak{F}$ and prove its termination, ultimately resulting in the decidability of our deductive system and assisting in our completeness result. Our approximation aims at producing a finite set of terms, such that we can then obtain any term of $\mathfrak{F}$ by simply applying functions to terms of the set, therefore producing a somewhat symbolic representation of the entire (infinite) term set $\mathfrak{F}$.

We begin by defining a function $\Rightarrow$ over sets of terms. This function takes a set and may add to the set a new term, if a reduction can be produced by placing terms from the set in a functional application context. This models the ability to apply functions to terms in order to reduce them to new terms.

The underlying idea is to understand a rewrite rule as a context where at the top level is a destructor function symbol, and under it lie contexts of possibly differing depths that consist solely of constructors. We then allow reductions to take place only in contexts such that the total context depth doesn't surpass the maximum possible depth of the instance of the rewrite rule. The motivation for this restriction is to prevent reductions from producing "redundant" terms in such a way that one could always add a new term to a set. We therefore allow a reduction to take place only when a "useful" term is being produced, in the sense that we are not simply adding terms of the initial set under functional contexts.

**Definition 3.3.4.** ***(Augmenting a set of terms).*** *Given a set of terms $S$, we define the procedure $S \Rightarrow S_1$ as the procedure that produces a set $S_1$ by extending $S$ with a term produced by the application of a single rewrite:*

$$S, t_1, \ldots, t_n \Rightarrow S, t_1, \ldots, t_n, M$$

$$\textit{if } \delta(g_1^{a_1}[t_1, \ldots, t_k], \ldots, g_i^{a_k}[t_{k'}, \ldots, t_n]) \to M, \forall i : a_i < K$$

*The several $g^i[\ldots]$ are contexts where constructors are applied to the terms $t_k$, with a total nesting depth of $i$. We require that the nesting doesn't surpass $K$, which is the sum of the maximum nesting in the rewrite rule for $\delta$ with the maximum nesting of the terms in $S$. This restriction introduces a bound on the size of terms that can be built, in order for the procedure to terminate.*

*We write $\Rightarrow^*$ to denote a transitive sequence of applications of $\Rightarrow$ and $\not\Rightarrow$ to denote that no more terms can be added by $\Rightarrow$.*

To clarify the nesting depth restriction imposed in the definition of $\Rightarrow$, consider a set of terms S such that:

$$S \triangleq \{\mathrm{enc}(x, y); y\}$$

We want our $\Rightarrow$ operation to be such that it will not continuously be able to add terms to the set. In this case, the term $x$ should be added, some other terms might be added as well, but a term such as enc(enc($x,y$), $y$) should not. Our definition of $\Rightarrow$ accurately accomplishes this by not allowing the context depths to grow indefinitely. By imposing a bound on the total nesting depth of the functional contexts of $\Rightarrow$, we only allow reductions up to the nesting depth where they can produce the relevant subterms from the initial set, that is, we only allow contexts up to the point where the deepest term from the initial set is placed in the deepest location of the rewrite rule.

We are now ready to define our approximation procedure for a set of terms $S$. The procedure blindly applies our $\Rightarrow$ operation assuming it will reach a saturation point, where no new terms can be added.

**Definition 3.3.5.** ***(Approximation of the Functional Equational Closure).*** *We define the set $b(S)$ as an approximation of the functional equational closure of the term $S$ as follows:*

$$b(S) = S' \triangleq S \Rightarrow^* S' \not\Rightarrow$$

It turns out to be the case that the previously mentioned saturation point is indeed always reached, as proved in Theorem 3.3.3. Intuitively, since our equations require the right hand side to be a subterm of the left, every reduction will always produce a smaller term, eventually reaching a point where no more reductions can be applied.

**Theorem 3.3.3.** ***(Termination of the Approximation Procedure).*** *Given any finite term set $S$, the computation of $b(S)$ always reaches a saturation point and hence always terminates.* Proof: *Appendix A.5*

In spite of our decidability result, it is still necessary to prove that given a set $S$, our approximation $b(S)$ is indeed such that it serves as a basis from which we can compute any term of $\mathfrak{F}$(S). What we require is that for every term in $\mathfrak{F}(S)$, we can generate the term by applying a functional context to terms of $b(S)$. It turns out that our approximation procedure is valid (shown in Theorem 3.3.4), which thus results in a set that does indeed symbolically represent the whole of $\mathfrak{F}$.

**Theorem 3.3.4.** ***(Validity of the Approximation Procedure).*** *Given a set of terms $S$, we have that:*

$$\forall M \in \mathfrak{F}(S), \exists\, C, \bar{t} \in b(S) : M = C[\bar{t}]$$

*Where $C$ is an arbitrary function application context.* Proof: *Appendix A.6*

We have thus determined the decidability of our proof system under some reasonable assumptions. Given two knowledge formulas $\varphi$ and $\psi$ such that we want to verify if $\varphi \vdash \psi$, we can compute the approximation of the functional equational closure of the terms denoted by $\varphi$ and then check if the terms denoted by $\psi$ can be constructed from those in the approximation, which is evidently a finite process.

However, our approximation also serves another purpose. We use it to also obtain the proof of completeness of our proof system. The proof is done in two steps. We first show our proof system to be complete w.r.t our approximation (which by itself is complete w.r.t to $\mathfrak{F}$) in Theorem 3.3.5, by showing that we can simulate the application of functional contexts and destructor application in our proof derivations. We then produce our main result in Theorem 3.3.6 by using Theorems 3.3.4 and 3.3.5 and showing that our proof system can simulate the necessary functional contexts.

**Theorem 3.3.5.** ***(Completeness of System** $K$ **w.r.t the Approximation Procedure).*** *Given a set of terms $S$, if $M \in b(S)$ then $S \vdash M$.* Proof: *Appendix A.7.*

**Theorem 3.3.6.** ***(Completeness of System** $K$ **w.r.t** $\mathfrak{F}$***).*** *Given the set of terms $\mathfrak{F}(S)$, if $M \in \mathfrak{F}(S)$ then $S \vdash M$.*

*Proof:* From Theorem 3.3.4 we have that if $M \in \mathfrak{F}(S)$ then $\exists\, C, \bar{t} \in b(S) : M = C[\bar{t}]$. Also, from Theorem 3.3.5 we know that $\forall t_i \in \bar{t} : S \vdash t_i$. To prove that $S \vdash M$, we need only be able to apply function symbols on the right of the derivation, which we do using rule (funRight):

$$\dfrac{\dfrac{S \vdash t_1}{\vdots} \qquad \cdots \qquad \dfrac{S \vdash t_n}{\vdots}}{S \vdash C[t_1, \ldots, t_n]}$$

We can simulate the functional structure of context $C[-]$ by repeatedly using the (funRight) rule to apply function symbols when needed, up to the various minimal $t_i$. □

With this we conclude our theoretical results that aim at validating our work. To summarise, we have shown our proof system for knowledge formulas to be both sound and complete with regard to its intended semantics (closure of term sets) and we proved it to be decidable, thus showing the semantics of our knowledge predicate to be decidable and therefore suitable to be implemented in a tool, as will be elaborated upon in the next section.

# 4. Extending the Spatial Logic Model Checker

In this chapter we detail the implementation of the logic and model of Chapter 3 into the Spatial Logic Model Checker tool. We begin by illustrating the tool itself and some of the internals of its original implementation. We then go over our implementation goals and identify the extension points in the implementation. Moving on to the implementation itself, we detail the relevant parts of our implementation and some of the challenges that were met. Finally, we present some detailed examples that illustrate the usage of the tool and the framework.

## 4.1 The Spatial Logic Model Checker: Initial Version

The SLMC is a model-checker that allows the verification of properties written in a spatial-behavioural logic over $\pi$-calculus processes [60]. The model checking algorithm was implemented using on-the-fly model checking techniques and written in OCaml. Broadly speaking, the original process language of the tool is similar to our process calculus of Section 3.1 but without structured terms. Therefore only allowing processes to communicate flat values. The logic of the tool is essentially our core spatial-behavioural logic (without connectives that relate to term reasoning).

### 4.1.1 Syntax

We now present the syntax of the original tool. We begin with the syntax of processes, which is a fairly straightforward representation of parametrized $\pi$-calculus polyadic processes. We then present the syntax of formulas of the spatial behavioural logic, and finally present some of the top level commands of the tool.

The processes are standard $\pi$-calculus processes with a test prefix, a $\tau$ prefix, guarded choice and recursive definitions. The concrete syntax of processes is defined in Figure 4.1. In this representation, an output prefix $x\langle y_1,\ldots,y_n\rangle$ is written as $x!(y_1,\ldots,y_n)$ and an input prefix $x(y_1,\ldots,y_n)$ is written as $x?(y_1,\ldots,y_n)$. The test prefix, that fires on name equality, is written as [*name* = *name*]. The $\tau$ prefix stands for an internal computation step. The *select* construct, which represents sumations, refers to alternative (guarded) branches, separated by a semicolon.

To allow recursive (and mutually recursive) definitions, the form *CapsId*(*namelist*) refers to a process identifier that can be defined using the top level command *defproc*. This command allows us to bind a parametric process definition to an identifier which can be used in process definitions.

The syntax of formulas is described in Figure 4.2. The boolean connectives are standard negation *not*, conjunction *and*, disjunction *or*, implication => and equivalence <=>. Spatial connectives are *void*, composition |, decomposition || (dual of composition), name revelation *reveal*, its dual *revealall* and the free-name occurrence connective @. Names can be quantified over universally (*forall*), existentially (*exists*) and we can quantify explicitly over fresh names

```
lower     ::= [‘a’ - ‘z’]
upper     ::= [‘A’ - ‘Z’]
letter    ::= lower | upper
digit     ::= [‘0’ - ‘9’]
name      ::= lower (letter | digit | ‘_’)*
namelist  ::= ϵ | name(‘,’ name)*
prefix    ::= name!(namelist)
           |  name?(namelist)
           |  [name = name]
           |  τ
process   ::= 0
           |  process | process
           |  new namelist in process
           |  prefix.process
           |  select {prefix.process (‘;’ prefix.process)*}
           |  CapsId(namelist)
           |  (process)
```

Figure 4.1: SLMC Process Syntax

```
formula  ::=  formula | formula
          |   formula || formula
          |   formula => formula
          |   formula <=> formula
          |   formula and formula
          |   formula textor formula
          |   (formula)
          |   not formula
          |   void
          |   true
          |   false
          |   name == name
          |   name! = name
          |   @ name
          |   exists name . formula
          |   forall name . formula
          |   reveal name . formula
          |   revealall name . formula
          |   hidden name . formula
          |   fresh name . formula
          |   ⟨label⟩formula
          |   [label]formula
          |   minfix CapsId.formula
          |   (minfix CapsId(namelist).formula)(namelist)
          |   maxfix CapsId.formula
          |   (maxfix CapsId(namelist).formula)(namelist)
          |   CapsId
          |   CapsId(namelist)
          |   k
          |   inside formula
          |   always formula
          |   eventually formula
          |   Id(namelist, formulalist)
label    ::=  tau
          |   name
          |   ?
          |   !
          |   name?
          |   name!
          |   name?(namelist)
          |   name!(namelist)
          |   *
```

Figure 4.2: SLMC Formula Syntax

(*fresh*) and restricted names (*hidden*). We can also test for name equality with == and ! =.

Behavioural modalities appear as $\langle label \rangle$ and $[label]$ expressing possibility of action and necessity of action, respectively. The *label* can be instantiated as an internal reduction step (*tau* or $\epsilon$), an input or an output on subject *name*, any input action ?, any output action !, any output action on subject *name* (*name*!), any input action on subject *name* (*name*?), a particular input action (*name*?(*namelist*)), a particular output action (*name*!(*namelist*)), or $*$ for any of the previously specified actions.

Recursive formulas can also be defined using the *minfix* and *maxfix* operators, that denote the least fix point and greatest fix point operators, respectively. The *k* construct, where *k* is an integer constant, denotes processes composed of *k* components. The *inside* construct allows for inspection of a formula under all name restrictions, with all the restrictions revealed using fresh names. Finally, the *always* construct can be read as 'for every possible configuration' and *eventually* as 'there will be a configuration', with regard to the system's internal evolution.

Similarly to process definitions, but not allowing for explicit recursion, a formula definition can be bound to a formula identifier using the top level *defprop* command.

### 4.1.2 Implementation Details

As previously mentioned, the SLMC tool uses on-the-fly model checking techniques. This entails that the state space of a process is not fully constructed at the start but is called upon as the model-checking procedure so requires. To this end, we will now overview the internal representation of processes and briefly describe how the model-checking procedure operates. For more on the specifics and theoretical aspects of the SLMC tool we refer to [11], where the model-checking algorithm was presented and proved decidable for a useful class of processes.

Internally, processes are represented in a normal form through an equation system. Every equation in such a system is identified by an equation variable. In this context, an equation models the set of all available top level actions of a process, each with a continuation that is identified by the adequate equation variable. An equation can therefore be read as a set of summations of actions, composed in parallel, under the necessary restricted names, where the continuations of each action are referenced by an equation variable. When a process is defined in the tool, it is transformed into this equational normal form and stored as such.

The model checking algorithm doesn't manipulate the equation systems directly, instead manipulating processes as sets of components generated from the equation system. A component is essentially an equation, containing top level actions where each continuation is referenced to by the appropriate equation variable. However, a component is defined by the restricted names of a process in the sense that each component represents a set of processes running in parallel that are indivisible due to their restricted names. For instance, the process `new x in c!(x) | d!(x)` would be represented by a single component that has two output actions, whereas the process `c!(x) | d!(x)` would be represented by two distinct components, one with the output action with subject *c* and the other with the output action with subject *d*.

```
deffun enc/2;
deffun pk/1;
defreduc dec(enc(x,y),pk(y)) = x;
defreduc dec(enc(x,pk(y)),y) = x;
```

Figure 4.3: Defining Equational Theories and Signatures

The model checking algorithm is driven by the formula that is to be checked, in the sense that it navigates the state space of the process as the formula so demands. This is accomplished via three kinds of process iterators: the action iterator, the revelation iterator and the composition iterator. Intuitively, the action iterator will take a process and an action and produce (at each iteration step) a process that results from the original process performing the supplied action. For example, if the considered action is an input and such an action is performable by the process, the tool will evolve the process by instantiating the input parameters as fresh names. The revelation iterator, given a process and a name, produces all resulting processes where the name has been revealed (i.e. taken as a fresh name). The composition iterator takes a process and returns all possible decompositions of that process.

The algorithm is thus defined inductively on the structure of the formula to be checked, using the appropriate iterators as needed. For quantification over names it uses the set of free names of the process plus a fresh name as the domain of the quantifier witness.

## 4.2 The Spatial Logic Model Checker: Our Extension

We will now detail the several extensions that were implemented on the SLMC tool. We begin by identifying the key features that needed to be added in order to support the new logic and model language. We then proceed by describing how such features were implemented.

In spite of all the modifications that needed to be implemented in order to successfully extend the SLMC tool, we wanted to keep the principal design philosophies of the tool: we wanted to maintain the internal representation of processes as equations, the components as minimal units under restricted names, the iterators to explore the state space of processes. Therefore, our extensions were designed with these goals in mind.

### 4.2.1 Syntax

First, we needed to adapt the $\pi$-calculus model of the tool to allow processes to handle not just flat names, but structured terms built up from functions of names. Coupled with this, we also needed to be able to define the semantics of such functions in order to precisely define our term reasoning.

To define our function signatures and equational theories, we introduce two new top level

```
              ...
term       ::=  name
             |  name(termlist)
termlist   ::=  ε | term(‘,’ term)*
prefix     ::=  name!(termlist)
             |  name?(namelist)
             |  [term = term]
             |  τ
process    ::=  let name = term in process
             |  ...
```

Figure 4.4: SLMC Extended Process Syntax

commands *deffun* and *defreduc*. The *deffun* command allows us to define constructors by specifying a function symbol and its arity. The *defreduc* command allows us to define destructors by specifying an equation where at the top level of the left hand side we have a destructor function symbol, applied to constructors and names, and on the right hand side we have a subterm of the l.h.s. For example, we can define the signature and equational theory for asymmetric cryptography with the commands of Figure 4.3, where we define a constructor `enc` which will model the encrypted terms, a constructor `pk` which models public keys and a destructor `dec` which models the asymmetric decryption of terms.

In Figure 4.4 we present the extended process syntax. In the implementation of our processes, we extended the output prefix by allowing structured terms to be output, we implemented our *let* declaration and we also implemented our attacker output of Section 3.1.4.

Regarding the logic, whose extended syntax is presented in Figure 4.5, we implemented a new action label that considers output of structured terms, a knowledge connective *knows* that allows us to phrase properties of the values a process can produce, given those it possesses and a *secret* quantifier, which allows us to quantify over terms that exist in a process and cannot have been produced by other processes (due to the presence of restricted values).

### 4.2.2 Internals

We will now overview how the several syntactical extensions to the tool are realised internally. Starting off with the representation of functions and equational theories, we simply store the declarations in an internal data structure and define our term rewriting system through a procedure that given a term, the signature and the sets of equations, generates the normal form of the given term by generating unifiers with the appropriate rewrite rules. With the term rewriting implemented, we then proceed to our process calculus model internals.

| | | |
|---|---|---|
| `formula` | ::= | … |
| | \| | *secret name* . `formula` |
| | \| | *knows* `knowledge_formula` |
| `knowledge_formula` | ::= | `knowledge_formula` *and* `knowledge_formula` |
| | \| | `term` |
| `label` | ::= | *tau* |
| | \| | *name* |
| | \| | ? |
| | \| | ! |
| | \| | *name*? |
| | \| | *name*! |
| | \| | *name*?(`namelist`) |
| | \| | *name*!(`termlist`) |
| | \| | * |

Figure 4.5: SLMC Extended Formula Syntax

#### 4.2.2.1 **Model**

In the input label prefix extension, we replaced the instantiation of the input parameters as fresh names by instantiation of the parameters as a fresh name or any of the free names that occur in the process. For the output prefix extension, we simply allow terms to occur in the process definitions and in the internal representations. We then modified the action iterator to only allow output prefixes to be fired if the term is valid (i.e. its normal form is destructor-free). For the *let* declaration we bind the declaration to the identifier, but only allow processes to proceed past the declaration if the declared value can be reduced to a destructor-free term. In the test prefix, we reduce both terms to a normal form and check if they are destructor-free and syntactically equal for the prefix to be fired.

Finally, to implement our attacker output prefix, we first implemented a relevant term extraction procedure similar to the specification of Definition 3.2.6 but that doesn't remove terms with restricted names (although it doesn't necessarily reveal them to the attacker). This modification to the term extraction procedure is required since the idea is that the attacker can output any term it can construct, including terms with restricted names. For instance, if the attacker receives a message cyphered with a key that is a restricted name, even though he cannot decrypt the message if he doesn't know the key, he can still forward it through a channel. Given this set of extracted terms, we then implemented a procedure that computes the approximation to the functional and equational closure of the set, following Definition 3.3.5. Since the definition of the attacker output prefix is that we allow the output of any terms in the functional equational closure, we would have to consider an infinite amount of possible outputs. To circumvent this, we define a top level parameter in the tool which we name *attacker depth*. This parameter defines the maximum function nesting depth for terms the attacker output prefix can indeed output. We then implemented a procedure that given an approximation of $\mathfrak{F}$, a signature $\Sigma$ and the *attacker depth* parameter $d$, computes the set of all terms that can be generated from the approximation by applying functions from $\Sigma$, with maximum nesting depth $d$. This parameter does indeed limit the power of our attackers, seeing how it is possible to have an undetected attack situation due to the size limit on the messages produced by attackers. However, we believe it to be possible to statically determine the maximum size of the attacker messages for a given protocol, which is expected from the work of Rusinowitch and Turuani [53], where they determine that the size of attacker messages is linear in the size of the protocol.

Lastly, we modified the action iterator to use the previous procedure to generate the terms that can be output in the attacker output prefix, which are then iterated over to navigate the state space accordingly.

#### 4.2.2.2 **Logic**

Regarding our internal implementation of the logical connectives, we implemented the new action label that considers output of terms via our revised action iterator (Section 4.2.2.1), where we match destructor-free normal forms of the objects of actions.

The *knows* connective is implemented by first extracting the relevant terms of the process according to Definition 3.2.6. Then, similar to what is done for the attacker output, we compute the approximation of the functional equational closure of the extracted set of terms (following Definition 3.3.5) and we then verify (per iteration of the approximation procedure for efficiency purposes), for each term in the knowledge formula, that we can generate such a term given those that figure in our approximation. This is achieved by iteratively decomposing each term from its functional context. For instance, if we obtain an approximation $b(S) = \{a; b\}$ and we wish to check if the term $f(h(a))$ can be generated from the approximation, we first check if $f(h(a))$ is in the approximation. Since that is not the case, we then check for $h(a)$ which is also not the case, for which we finally check for $a$. Since $a$ is in our approximation then the term $f(h(a))$ can be known (constructed) by a process who knows $a$ and $b$.

The *secret* quantifier is implemented straightforwardly using our term extraction procedures and the revelation iterator. We use the iterator to reveal an hidden name, extract the terms of the underlying process, compute the approximation and filter out the terms where the hidden name does not occur. We then check the quantified formula by instantiating the witness with each possible term and repeat the process for each restricted name, as needed.

With this we conclude our implementation details for the extension of the SLMC tool. We do note that some of the techniques we use are proof of concept and aren't appropriately optimised, namely our approximation generation procedure. For this we implemented a preliminary version of a caching system, that stores approximations for previously computed term sets.

```
deffun enc/2;
defreduc dec(enc(x,y),y) = x;
```

Figure 4.6: Symmetric Cryptography in the SLMC

However, since our approximation procedure may not be fully executed (since we perform our verifications per iteration), we store not only full approximations but also partial approximations, which can be re-used in other computations.

## 4.3 Examples

We will now present two examples of protocols modelled in the tool with the intent of illustrating its usage and as a way of validating the implementation itself. First, we'll re-introduce the example of Section 3.1.5, using the syntax of the tool, as an example of a simple toy protocol which we verify both standalone (using behavioural properties) and in the presence of attackers, using spatial, behavioural and knowledge properties. We then proceed to a more complex example, which consists of the Needham-Schroeder protocol with asymmetric cryptography [49]. This protocol is a widely known mutual authentication protocol that was found to have a severe vulnerability [42]. The exploit was such that it was possible for an attacker to falsely authenticate himself as one of the principals of the protocol. We will therefore explain and model the Needham-Schroeder protocol and verify that indeed such a vulnerability can be detected by our tool.

### 4.3.1 A Simple Protocol

Recalling on the example of Section 3.1.5, we have a protocol where two principals, Alice and Bob, share a symmetric key and wish to agree on a session key they can use to secure their communications. We then want to verify that the protocol is correct with and without the presence of attackers.

We begin by modelling the allowed cryptographic operations in our protocol, which in this simple case consists solely of symmetric cryptography. We model encryption with a constructor `enc` of arity two, and decryption with a destructor `dec`, that receives an encrypted value and reveals it if the keys match, as defined in Figure 4.6.

We now define our two principals, Alice and Bob. Alice will await for a message `m` over channel `c`, which will then be decrypted using the shared key `k`. The idea is that this encrypted message will be emitted by Bob, and will contain the session key. In order for Alice to confirm that she received the message she will then encrypt the value `hello` using the session key and send it to Bob, over channel `c`, and simply repeat her behaviour.

Bob behaves symmetrically to Alice. He begins by generating the session key `key`, encrypting it using the shared key `k` and sending it to Alice over channel `c`. He will then await for

```
defproc Alice(k) = c?(m).
                   let sessionKey = dec(m,k) in
                   c!(enc(hello,sessionKey)).Alice(k);

defproc Bob(k) = new key in
                 c!(enc(key,k)).
                 c?(x).
                 [dec(x,key)=hello].
                 ok!().
                 Bob(k);

defproc System = new k in
                 (Alice(k) | Bob(k));
```

Figure 4.7: A Simple Protocol in the SLMC

```
check System |= always eventually <ok!> true;

* Process System satisfies the formula
                                  always (eventually (<ok!> (true))) *
```

Figure 4.8: Local Correctness

Alice's reply and check that the received message is indeed the value `hello` encrypted with the session key. If such is the case then Bob will emit an ok signal, modelled by an output on channel `ok`, and repeat his behaviour.

Our protocol is therefore composed by Alice and Bob running in parallel, both sharing a symmetric key, as shown in Figure 4.7.

We can now verify that the message exchanges that make up the protocol are correct without the presence of attackers, in the sense that Alice and Bob can always, after exchanging the necessary messages, agree on the session key. Since when the session key is agreed Bob outputs on `ok`, we can check that for every configuration of our system, we will reach a configuration where we can observe the `ok` signal. As shown in Figure 4.8, such is indeed the case.

We now move for the verifications in the presence of attackers. For this we need to model an attacker for the protocol, which can be done following the schema of Session 3.1.4 for active attackers, and represent our "World", which consists of the initial system and the attacker, defined in Figure 4.9. Our attacker behaves as expected, intercepting the first message of the protocol, injecting any message it can produce using our notion of knowledge and repeat this behaviour for the second message of the protocol. The last action acts as a memory, so the intercepted messages persist throughout the life cycle of the attacker.

```
defproc Attacker = c?(x).
                   c!(*/1).
                   c?(y).
                   c!(*/1).
                   store!(x,y);

defproc World = (System | Attacker);
```

Figure 4.9: The World and the Attacker

```
defprop threeHellos = eventually inside
                      (knows hello | knows hello | knows hello);
check World |= always not threeHellos;

* Process World satisfies the formula always (not threeHellos) *
```

Figure 4.10: Verification of the *hello* value

To ensure our toy protocol is correct, we now perform two verifications. First we check that the attacker cannot obtain the `hello` value that is exchanged between Alice and Bob. We do this by expressing an internal configuration of our world where three agents know `hello`, our two principals and the attacker. Since we do not want such a configuration to be reachable, we check that indeed such is the case, as shown in Figure 4.10.

The second verification consists of checking that the attacker doesn't obtain either the session key or the shared key. For this we begin by defining two auxiliary properties for readability purposes (Fig. 4.11), that model logic conjunction and disjunction over knowledge.

We then express our undesired configuration by quantifying over our restricted names (the keys) and stating that, under the restrictions, we can reach a state where we have two principals that know both keys (which are Alice and Bob) and one that knows one or the other (the attacker). Finally, we verify that such is not the case, as shown in Figure 4.12. Since we know that Alice and Bob do obtain the keys, we know the attacker cannot obtain either, and thus conclude our verification of our toy example.

### 4.3.2 The Needham-Schroeder Protocol

The Needham-Schroeder (asymmetric cryptography) protocol is a standard mutual authentication protocol in distributed systems. Its goal is to have two principals authenticate each other (using asymmetric cryptography) with the help of a trusted third party, which acts as a server that distributes public keys on request.

The protocol consists of seven messages and we assume that each principal only knows its

```
defprop kAnd(v1,v2) = knows (v1 and v2);
defprop kOr(v1,v2) = knows v1 or knows v2;
```

Figure 4.11: Auxiliary Properties

```
defprop badConfig = hidden k1.hidden k2.
        (eventually (kAnd(k1,k2) | kAnd(k1,k2) | kOr(k1,k2)));
defprop noKeyLeak = not badConfig;
check World |= noKeyLeak;

* Process World satisfies the formula noKeyLeak *
```

Figure 4.12: Key Leaking Absence

$$\begin{array}{lcl}
A \rightarrow S & : & A,B \\
S \rightarrow A & : & \{K_{PB},B\}_{K_{SS}} \\
A \rightarrow B & : & \{N_A,A\}_{K_{PB}} \\
B \rightarrow S & : & B,A \\
S \rightarrow B & : & \{K_{PA},A\}_{K_{SS}} \\
B \rightarrow A & : & \{N_A,N_B\}_{K_{PA}} \\
A \rightarrow B & : & \{N_B\}_{K_{PB}}
\end{array}$$

Figure 4.13: The Needham-Schroeder Protocol

own secret key plus the public key of the trusted server, which can be used to request any public key in the system. Using the standard Alice and Bob naming schemes, Alice wants to establish a mutually authenticated connection with Bob. It begins by requesting Bob's public key from the trusted server. The server sends Bob's public key in a message signed with its own secret key. Alice will then receive Bob's key from the server and use it to encrypt a message that contains Alice's identity and a nonce that ensures the freshness of the message. Any agent can generate such a message, but since its encrypted with Bob's public key, Alice knows that only Bob can decrypt it. Bob will receive the message and extract the nonce and Alice's identity from it. Bob will then use Alice's identity to query the server for Alice's key. After Bob receives Alice's key he will generate a fresh nonce and send Alice's original nonce and the new one, encrypted with Alice's key. This message proves that Bob successfully decrypted Alice's message.

When Alice receives the message, she'll extract the two nonces and compare the nonce she initially sent with the one Bob believes Alice sent. If they match, Alice will reply with Bob's nonce encrypted with Bob's public key. The nonce test checks the freshness of the message sequence, while the reply serves as proof for Bob that Alice did receive his nonce. After Bob receives the final message, he'll test the received nonce against his own, and if there is a match both principals will have successfully authenticated each other in such a way that the two nonces are known only by them. We present the message exchanges of the protocol in Figure 4.13. In the diagram, A stands for Alice, B for Bob and S for the Server. $K_{P-}$ and $K_{S-}$ stand for public and secret keys, respectively. $N_A$ is a fresh value (nonce) generated by Alice and $N_B$ is a nonce generated by Bob.

However, his protocol is vulnerable to a man-in-the-middle attack. If some attacker can persuade Alice to initiate a session with him, he can replay the messages to Bob in such a way that at the end of the attack, Bob will falsely believe that Alice is communicating with him and that $N_A$ and $N_B$ are known only to him and Alice. We will now model such an attack.

#### 4.3.2.1 **The Man-in-the-Middle Attack**

For the attack to take place, the attacker must first convince Alice to initiate a session with him. This means that in this attack, the attacker must be a member of the system - the server must have a public key for the attacker. To model these assumptions, we will model our attacker as a member of the system, and Alice as a process that awaits for a message that contains the identifier of the agent whom Alice is supposed to initiate a session with.

The message sequence of the attack is displayed in Figure 4.14, not considering the communications to and from the server which remain unchanged. In the message sequence, $T$ represents the malicious system member, Trudy.

After the second message, Trudy will know Alice's nonce, and use this information to falsely convince Bob that he's communicating with Alice. After the fifth message, Trudy will know Bob's nonce, and after sending the sixth message will successfully convince Bob that he is communicating with Alice.

We will now use the SLMC to verify that such an attack is indeed possible. We begin with

$$\begin{array}{rcl} A \rightarrow T & : & \{N_A, A\}_{K_{PT}} \\ T \rightarrow B & : & \{N_A, A\}_{K_{PB}} \\ B \rightarrow T & : & \{N_A, N_B\}_{K_{PA}} \\ T \rightarrow A & : & \{N_A, N_B\}_{K_{PA}} \\ A \rightarrow T & : & \{N_B\}_{K_{PI}} \\ T \rightarrow B & : & \{N_B\}_{K_{PB}} \end{array}$$

Figure 4.14: Attack Message Sequence

```
deffun enc/2;
deffun pk/1;
defreduc dec(enc(x,pk(y)),y) = x;

deffun sign/2;
defreduc sigcheck(sign(x,y),pk(y)) = x;

deffun pair/2;
defreduc px(pair(x,y)) = x;
defreduc py(pair(x,y)) = y;
```

Figure 4.15: Needham-Schroeder Signature and Eq. Theory

the necessary signature and equational theory for asymmetric encryption, message signatures and pairing, as defined in Figure 4.15.

We can now define Alice and Bob. For simplicity, and since the server communications are not considered for the attack, we will use a public channel `servchan` for communicating with the server and a public channel `c`. Both Alice and Bob behave accordingly to the message sequence chart of Figure 4.13, with the exception that Alice will first receive the identifier of the user she should initiate a session with. This idea is that this step will simulate the attacker convincing Alice to initiate a session with him. Figure 4.16 shows Alice and Bob's process code, where both agents know the server's public key and their own secret key. When Alice believes to have authenticated with a valid principal, she will emit an ok signal by outputting on channel `okAlice`. Bob will do the same over channel `okBob`.

The server, defined in Figure 4.17, waits for queries on channel `servchan` and replies with the appropriate signed key. We model a server who explicitly holds keys for Alice (the `[y=a]` branch), for Bob (the `[y=b]` branch) and for the attacker (the `[y=t]` branch).

We can now define the attacker (Fig. 4.18). It will begin by obtaining all the public information of the system: the public keys of Alice, Bob and the Server. It will then initiate the attack

```
defproc Alice(pubServer,secretA) =
     c?(host).
     servchan!(a,host).
     servchan?(x).
     let pubh = px(sigcheck(x,pubServer)) in
     new nonceA in
     c!(enc(pair(nonceA,a),pubh)).
     c?(y).
     let nA = px(dec(y,secretA)) in
     let nX = py(dec(y,secretA)) in
     [nonceA = nA].
     c!(enc(nX,pubh)).
     okAlice!();

defproc Bob(pubServer,secretB) =
      c?(x).
      let nY = px(dec(x,secretB)) in
      let host = py(dec(x,secretB)) in
      servchan!(b,host).
      servchan?(y).
      let pubh = px(sigcheck(y,pubServer)) in
      new nonceB in
      c!(enc(pair(nY,nonceB),pubh)).
      c?(z).
      let nB = dec(z,secretB) in
      [nonceB = nB].
      okBob!();
```

Figure 4.16: Needham-Schroeder Protocol: Bob and Alice

```
defproc Server(pubK_A,pubK_B,pubK_T,secretK) =
    servchan?(x,y).
    select{
     [y=b].
     servchan!(sign(pair(pubK_B,y),secretK)).
     Server(pubK_A,pubK_B,pubK_T,secretK);
     [y=a].
     servchan!(sign(pair(pubK_A,y),secretK)).
     Server(pubK_A,pubK_B,pubK_T,secretK);
     [y=t].
     servchan!(sign(pair(pubK_T,y),secretK)).
     Server(pubK_A,pubK_B,pubK_T,secretK)
    };
```

Figure 4.17: Needham-Schroeder Protocol: Server

by sending its identifier `t` to Alice, in order to force Alice to begin a session with him. From this point on it behaves as a regular attacker, intercepting messages and injecting any message it can produce in the protocol. If the attacker reaches the end of its message sequence, it will signal this fact using channel `okTrudy`. This will mean that our attacker will have successfully authenticated himself (as Alice) to Bob.

Since our attacker is a member of the system, we must model our system accordingly (Fig. 4.19). We therefore define our system by generating the secret keys for the attacker, Alice, Bob and the Server. We bootstrap the attack by sharing the public keys of Alice, Bob and the Server with the attacker, which is running in parallel with Alice, Bob and the Server.

Finally, we can now verify that the protocol does indeed have an attack, by checking if we eventually reach a configuration where Alice, Bob and our attacker have signalled their authentications as successful. If the attacker signals `okTrudy`, it will mean that it has obtained both the protocol nonces and has finished his attempt of leading Bob to believe the attacker is in fact Alice. The `okAlice` signal will mean that Alice has finished her authentication run with the attacker, therefore having sent her nonce and the nonce she believes was generated by the attacker (which was in fact generated by Bob). At last, the `okBob` signal will be given when Bob believes to have authenticated Alice, after having received the appropriate nonce. The verification of the attack is given in Fig. 4.20.

This vulnerability can be fixed by changing message number six in the protocol sequence. That is, replace the message $\{N_A, N_B\}_{K_{PA}}$ with:

$$B \rightarrow A \quad : \quad \{N_A, N_B, B\}_{K_{PA}}$$

By adding Bob's identity to the message, the attacker can no longer trick Alice into a wrongful authentication. We refrain from presenting the modelling of the fixed version of the protocol,

```
defproc Attacker(secretT) =
    c?(pkK,pkA,pkB).
    c!(t).
    c?(m).
    c!(*/1).
    c?(nounces).
    c!(*/1).
    c?(n).
    c!(*/1).
    okTrudy!(secretT,pkB,m,nounces,n);
```

Figure 4.18: Needham-Schroeder Protocol: Attacker

```
defproc Sys =
    new secretA,secretB,secretT,secretK in
    (c!(pk(secretK),pk(secretA),pk(secretB)).
    (Bob(pk(secretK),secretB) |
    Alice(pk(secretK),secretA) |
    Server(pk(secretA),pk(secretB),pk(secretT),secretK)) |
    Attacker(secretT));
```

Figure 4.19: Needham-Schroeder Protocol: System

```
defprop attack = eventually
     (<okTrudy!> true | <okBob!> true | <okAlice!> true);
check Sys |= attack;

- Time elapsed: 73.864063 secs -

- Number of state visits: 42715 -

* Process Sys satisfies the formula attack *
```

Figure 4.20: The Attack

```
defprop attack = eventually
     (<okTrudy!> true | <okBob!> true | <okAlice!> true);
check FixedSys |= attack;

- Time elapsed: 182.327999 secs -

- Number of state visits: 39635 -

* Process Sys does not satisfy the formula attack *
```

Figure 4.21: Fixing the Attack

for briefness purposes, but assert that indeed the fixed version does not have the previously described vulnerability (Fig. 4.21).

And so we end our example section, by showing analysis techniques for both small and larger protocols.

# 5 . Closing Remarks

Recalling our main goal of developing a logic suitable for security protocol analysis, we believe to have fulfilled this goal through the development of our spatial-behavioural-epistemic logic. Our logic is adequate for protocol reasoning since not only does it allow us to phrase interesting properties of protocols in terms of space, behaviour and knowledge, but it does this while remaining decidable, which then allowed us to develop a model checker for the logic, thus realising our initial goal. The development of the logic itself was not without challenges, of which we single out the development of our proof system that despite being composed of a small number of rules, remains powerful enough to preserve completeness. Perhaps the greatest challenges of our work were the technical results, specifically the completeness and decidability results, since not only were these essential in the face of our model checking goals, but also due to the non-trivial nature of such results.

We will now summarise the full scope of the work that ultimately resulted in this document. We started by performing a fairly extensive study and presentation of the existing work on concurrency and distributed system analysis using formal methodologies, and its applications to the field of security protocol verification (Chapter 2). We contributed to this area by establishing a process calculus model (Section 3.1), inspired on the existing Applied $\pi$-calculus, that allows us to represent security protocols as sets of message-passing agents. The agents can then employ cryptographic techniques to implement the desired protocols that are not built-in but instead modelled explicitly within the calculus, through sets of equations that define the (symbolic) semantics of the desired operations. We also define attackers explicitly within the model, by allowing processes to perform a "special" output operation, that non-deterministically outputs any value the process can generate.

Over this model, we began with a core spatial-behavioural logic and extended it with epistemic flavoured connectives (Section 3.2) . These connectives allow reasoning about processes at the term level, by permitting us to analyse what values a process can produce by applying any available operations (such as decryption) over the values it possesses. These three degrees of reasoning: Spatial, behavioural and epistemic; supply a heightened expressive power that can be used to phrase interesting properties of security protocols, not only regarding their local correctness (i.e. without the presence of malicious agents) but also properties concerning the correctness of protocols when faced against certain types of attackers, such as Dolev-Yao attackers.

Regarding the logic, we precisely defined its syntax and denotational semantics and furthered the logic by developing a formal theory of knowledge deduction, embodied in a sequent calculus formulation that is detailed in Section 3.3. Our proof system gives a syntactical approach to the semantic-based notion of derivability of information (the ability to compute new values from existing ones), and we validated the approach by proving it to be both sound and complete regarding its intended semantic model.

Furthermore, we proved the existence of an algorithm which allows us to generate a symbolic (finite) representation of the full (infinite) set of derivable information of a given set of values, and therefore allows us to decide if a certain value can be constructed given a set of base values. With this result, we produced a proof-of-concept implementation of a model checking algorithm for our model and logic, as an extension of the Spatial Logic Model Checker tool (Chapter 4). We validated our proof of concept implementation by modeling and verifying small toy protocols and by identifying the known man-in-the-middle attack on the Needham-Schroeder protocol, as detailed in Section 4.3.

From a critical perspective, our explicit modeling of attackers might seem impractical. However, following the schemas outlined in Section 3.1.4 we can define a fairly standard form for attacker representation that is fairly independent from the specifics of the attacked protocol. Additionally, when dealing with more complex attack scenarios, like that of Section 4.3.2, it becomes inevitable that some additional steps have to be added to the protocol for the attack to be detected, even when attackers are represented implicitly by the environment (like in [7]). In our case, we had to model the attacker as a member of the system itself, and manually trigger the authentication of a principal with the attacker. However, from that point onwards, our attacker follows our generic schema, which further strengthens our belief that such explicit modeling is not too cumbersome and can shed some light on how attacks can take place.

Critically, our attacker depth parameter is indeed a limitation of our tool (more-so then of the framework itself), but, as we have mentioned, following [53], we hope to be able to statically determine the maximum required depth for attacker messages in future work.

Another possible critique of our work is that our implementation, being simply a proof-of-concept, is perhaps too inefficient for larger protocols. However, our aim was primarily to show that our logic could be used in a tool for protocol analysis. A goal which we believe to have achieved, despite not having considered more sophisticated optimisation techniques. Such optimisations fall under the scope of future work, which we will now briefly overview.

## 5.1 Future Work

An obvious follow up work to this dissertation would be to study the complexity of the algorithms we implemented upon our SLMC extension and then research possible optimisations to the algorithms with the goal of making the tool more robust and capable of handling larger protocols. An optimisation possibility would be to research a symbolic approach to the term sets, in order to minimise the term and state-space generation.

We believe that it would also be interesting to study the expressive power of the logic and the model, by developing process equivalence techniques similar to those of the applied $\pi$-calculus (eg. static equivalence, behavioural equivalence) and then frame the process equivalence induced by our logic with these equivalence relations.

On a broader scope, it would also be interesting to research other techniques for security reasoning, such as type and effect systems or static analysis techniques.

# A . Selected Proofs

## A.1 Proof of Lemma 3.2.1

For all terms $t$, $sub(t) = \varphi$ is such that $\varphi$ does not contain terms with destructor function symbols.
*Proof:* Induction on the derivation of $sub(t)$ and case analysis on the structure of the term.

1. Case $sub(n) = n$
   Trivial, since $n$ is a name.

2. Case $sub(f(t_1,\dots,t_n)) = sub(t_1) \wedge \cdots \wedge sub(t_n)$ with $f$ a destructor.
   From the I.H the several $sub(t_i)$ do not contain destructors. Since $f$ is discarded, we are done.

3. Case $sub(f(t_1,\dots,t_n)) = f(t_1,\dots,t_n)$ with $f$ a constructor and no $t_i$ contains name variables or destructors.
   Trivial from the fact that $f(t_1,\dots,t_n)$ is destructor-free.

4. Case $sub(f(t_1,\dots,t_n)) = sub(t_1) \wedge \cdots \wedge sub(t_n)$ with $f$ a constructor and some $t_i$ contains a name variable or a destructor.
   From the I.H the several $sub(t_i)$ do not contain destructors, so we're done.

□

## A.2 Proof of Theorem 3.2.1

Given a process $P$ such that $P \vdash_k \varphi$, the knowledge formula $\varphi$ is made up of terms that are values, that is, composed solely of names and/or constructor function symbols.
*Proof:* Induction on the derivation of $P \vdash_k \varphi$ and case analysis on the last step of the derivation.

1. Cases: $\mathbf{0} \vdash_k \top$ and $X \vdash_k \top$
   Trivially satisfied.

2. Case Sum

$$\frac{P \vdash_k \varphi \qquad Q \vdash_k \psi}{P + Q \vdash_k (\varphi \wedge \psi)}$$

   From the I.H, $\varphi$ and $\psi$ are made up of values, so we're done.

3. Case Parallel Composition

$$\frac{P \vdash_k \varphi \qquad Q \vdash_k \psi}{P|Q \vdash_k (\varphi \wedge \psi)}$$

From the I.H, $\varphi$ and $\psi$ are made up of values, so we're done.

4. Case Input

$$\frac{P \vdash_k \varphi}{n(x).P \vdash_k \varphi \uparrow x}$$

From the I.H, $\varphi$ is made up of names and constructor terms. The $\uparrow$ operation removes terms with newly identified variable $x$ from $\varphi$ but does not add destructors, therefore we're done.

5. Case Output

$$\frac{P \vdash_k \varphi}{x\langle M\rangle.P \vdash_k \varphi \wedge sub(M)}$$

From the I.H, $\varphi$ is made up of names and constructor terms. From Lemma 3.2.1, $sub(M)$ is destructor-free, so we're done.

6. Case Restriction

$$\frac{P \vdash_k \varphi}{(\nu n)P \vdash_k \varphi \uparrow n}$$

From the I.H, $\varphi$ is made up of names and constructor terms. The $\uparrow$ operation removes terms with the newly identified bound name $n$ from $\varphi$ but does not add destructors, therefore we're done.

7. Case Let

$$\frac{P\{n \leftarrow M\} \vdash_k \varphi}{\text{let } n = M \text{ in } P \vdash_k \varphi \wedge sub(M)}$$

From the I.H, $\varphi$ is made up of names and constructor terms. From Lemma 3.2.1, $sub(M)$ is destructor-free, so we're done.

8. Case Test

$$\frac{P \vdash_k \varphi}{[M = N].P \vdash_k \varphi \wedge sub(M) \wedge sub(N)}$$

From the I.H, $\varphi$ is made up of names and constructor terms. From Lemma 3.2.1, *sub*($M$) and *sub*($N$) are destructor-free, so we're done.

9. Case Recursive Def.

$$\frac{P \vdash_k \varphi}{(\textbf{rec}\ X.P) \vdash_k \varphi}$$

Trivial from the I.H.

□

## A.3 Proof of Lemma 3.3.3

If a sequent $\Gamma \vdash A$ has a single cut derivation in *KC* then it has a cut-free derivation in KC.
*Proof:* Induction on the structure of the derivation of $\Gamma \vdash A$. By the assumptions of the lemma we have that the derivation is of the form

$$\frac{\dfrac{\Pi(n)}{\Gamma \vdash A} \qquad \dfrac{\Pi(m)}{\Gamma, A \vdash C}}{\Gamma \vdash C}\ \text{(Cut)}$$

where $\Pi(n)$ and $\Pi(m)$ are cut-free derivations for $\Gamma \vdash A$ and $\Gamma, A \vdash C$, respectively. The various forms of the premise derivations $\Pi(n)$ and $\Pi(m)$ fall under the following cases:

1. One of the premises is an instance of (Id).
2. One of the premises is an instance of a rule that doesn't introduce the cut formula.
3. Both premises introduce the cut formula

We now discuss the identified cases:

1. **Case (Cut)-(Id):** Suppose (Id) occurs on the right premise of the cut, the derivation must have the form

$$\frac{\dfrac{\Pi(n)}{\Gamma \vdash A} \qquad \dfrac{}{\Gamma, A \vdash A}\ \text{(Id)}}{\Gamma \vdash A}\ \text{(Cut)}$$

and we can remove the cut since $\Gamma \vdash A$ follows from $\Pi(n)$, which is cut free. The case where (Id) appears on the left premise of the cut is handled symmetrically.

2. **Case (Cut)-(LL):** The cases where the left premise of the cut is a left rule that doesn't introduce the cut formula.

(a) Case (Cut)-($\wedge$L)

$$\dfrac{\dfrac{\dfrac{\Pi(n)}{\Gamma, A, B \vdash C}}{\Gamma, A \wedge B \vdash C}\,(\wedge\text{L}) \qquad \dfrac{\Pi(m)}{\Gamma, A \wedge B, C \vdash D}}{\Gamma, A \wedge B \vdash D}\,(\text{Cut})$$

Applying weakening on $\Pi(n)$ and $\Pi(m)$, we construct $\Pi'(n)$ and $\Pi'(m)$:

$$\dfrac{\dfrac{\Pi'(n)}{\Gamma, A, B, A \wedge B \vdash C} \qquad \dfrac{\Pi'(m)}{\Gamma, A, B, A \wedge B, C \vdash D}}{\Gamma, A, B, A \wedge B \vdash D}$$

By the induction hypothesis we have a cut-free derivation for $\Gamma, A, B, A \wedge B \vdash D$ and:

$$\dfrac{\dfrac{\vdots}{\Gamma, A, B, A \wedge B \vdash D}}{\Gamma, A \wedge B \vdash D}\,(\wedge\text{L})$$

since our sequents have formula sets (and not multi-sets) on the left-hand side, we have a derivation for $\Gamma, A \wedge B \vdash D$ cut-free.

(b) Case (Cut)-(DestrLeft)

$$\dfrac{\dfrac{\dfrac{\Pi(n)}{\Gamma, s \vdash C}}{\Gamma, f(\bar{t}) \vdash C}\,(\text{DestrLeft}) \qquad \dfrac{\Pi(m)}{\Gamma, f(\bar{t}), C \vdash D}}{\Gamma, f(\bar{t}) \vdash D}\,(\text{Cut})$$

Applying weakening on $\Pi(n)$ and $\Pi(m)$, we construct $\Pi'(n)$ and $\Pi'(m)$:

$$\dfrac{\dfrac{\Pi'(n)}{\Gamma, s, f(\bar{t}) \vdash C} \qquad \dfrac{\Pi'(m)}{\Gamma, f(\bar{t}), s, C \vdash D}}{\Gamma, s, f(\bar{t}) \vdash D}$$

By the induction hypothesis we have a cut-free derivation for $\Gamma, s, f(\bar{t}) \vdash D$ and:

$$\dfrac{\dfrac{\vdots}{\Gamma, s, f(\bar{t}) \vdash D}}{\Gamma, f(\bar{t}) \vdash D}\,(\text{DestrLeft})$$

since our sequents have formula sets (and not multi-sets) on the left-hand side, we have a derivation for $\Gamma, f(\bar{t}) \vdash D$ cut-free.

(c) Case (Cut)-(AttLeft)

$$\dfrac{\dfrac{\dfrac{\Pi(n)}{\Gamma, f(\bar{t}) \vdash C}}{\Gamma, \bar{t} \vdash C}\,(\text{AttLeft}) \qquad \dfrac{\Pi(m)}{\Gamma, \bar{t}, C \vdash D}}{\Gamma, \bar{t} \vdash D}\,(\text{Cut})$$

Applying weakening on $\Pi(n)$ and $\Pi(m)$, we construct $\Pi'(n)$ and $\Pi'(m)$:

$$\dfrac{\dfrac{\Pi'(n)}{\Gamma, f(\bar{t}), \bar{t} \vdash C} \qquad \dfrac{\Pi'(m)}{\Gamma, f(\bar{t}), \bar{t}, C \vdash D}}{\Gamma, f(\bar{t}), \bar{t} \vdash D}$$

By the induction hypothesis we have a cut-free derivation for $\Gamma, f(\bar{t}), \bar{t} \vdash D$ and:

$$\dfrac{\dfrac{\vdots}{\Gamma, f(\bar{t}), \bar{t} \vdash D}}{\Gamma, f(\bar{t}) \vdash D}\,(\text{AttLeft})$$

since our sequents have formula sets (and not multi-sets) on the left-hand side, we have a derivation for $\Gamma, f(\bar{t}) \vdash D$ cut-free.

(d) Case (Cut)-(RL): The cases where the right premise of the cut is a left rule that doesn't introduce the cut formula are handled symmetrically to Case 2.

3. Cases where the cut premisses introduce the cut formula. The last rule on the left premise is a right rule and the last rule on the right premise is a left rule.

(a) Case of $\wedge$:

$$\dfrac{\dfrac{\dfrac{\Pi(n)}{\Gamma \vdash A} \quad \dfrac{\Pi(m)}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\,(\wedge\text{R}) \qquad \dfrac{\dfrac{\Pi(k)}{\Gamma, A, B \vdash C}}{\Gamma, A \wedge B \vdash C}\,(\wedge\text{L})}{\Gamma \vdash C}\,(\text{Cut})$$

From weakening we have:

$$\dfrac{\Pi'(n)}{\Gamma, B \vdash A} \qquad \dfrac{\Pi'(m)}{\Gamma, A \vdash B}$$

We can thus build the derivations:

$$\dfrac{\dfrac{\Pi'(n)}{\Gamma, B \vdash A} \qquad \dfrac{\Pi(k)}{\Gamma, A, B \vdash C}}{\Gamma, B \vdash C}\,(\text{Cut}) \qquad \dfrac{\dfrac{\Pi'(m)}{\Gamma, A \vdash B} \qquad \dfrac{\Pi(k)}{\Gamma, A, B \vdash C}}{\Gamma, A \vdash C}\,(\text{Cut})$$

and by the induction hypothesis we have cut-free derivations for $\Gamma, B \vdash C$ and $\Gamma, A \vdash C$. We then use such cut-free derivations to construct:

$$\dfrac{\dfrac{\Pi(n)}{\Gamma \vdash A} \qquad \dfrac{\vdots}{\Gamma, A \vdash C}}{\Gamma \vdash C} \quad \text{or} \quad \dfrac{\dfrac{\Pi(m)}{\Gamma \vdash B} \qquad \dfrac{\vdots}{\Gamma, B \vdash C}}{\Gamma \vdash C}$$

And by the induction hypothesis, there exist cut-free derivations for both sequents.

(b) Term formulae (funRight + DestrLeft)

$$\dfrac{\dfrac{\dfrac{\Pi(m_1)}{\Gamma \vdash t_1} \quad \dots \quad \dfrac{\Pi(m_n)}{\Gamma \vdash t_n}}{\Gamma \vdash f(t_1,\dots,t_n)}\,\text{(FunRight)} \qquad \dfrac{\dfrac{\Pi(k)}{\Gamma, t \vdash C}}{\Gamma, f(t_1,\dots,t_n) \vdash C}\,\text{(AttLeft)}}{\Gamma \vdash C}\,\text{(Cut)}$$

From the $\Pi(m_i)$ derivations we can build:

$$\dfrac{\dfrac{\dfrac{\Pi(m_1)}{\Gamma \vdash t_1} \quad \dots \quad \dfrac{\Pi(m_n)}{\Gamma \vdash t_n}}{\Gamma \vdash \bar{t}} \qquad \dfrac{\dfrac{\Gamma, t \vdash t}{\Gamma, f(\bar{t}) \vdash t}}{\Gamma, \bar{t} \vdash t}}{\Gamma \vdash t}\,\text{(Cut)}$$

From the induction hypothesis we can derive $\Gamma \vdash t$ cut-free. Finally:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t} \qquad \dfrac{\Pi(k)}{\Gamma, t \vdash C}}{\Gamma \vdash C}\,\text{(Cut)}$$

And by application of the induction hypothesis there exists a cut-free derivation for $\Gamma \vdash C$. □

## A.4 Proof of Theorem 3.3.2

If $\Gamma \vdash A$ is derivable in the proof system K, then $\Gamma \models A$.
*Proof:* Induction on the derivation of $\varphi \vdash \psi$
We assume $\varphi$ to be of the form $v_1, \dots, v_k$.

1. Cases (Id) and ($\wedge$: left)

   Trivial from the I.H.

2. Case ($\wedge$: right)

$$\dfrac{\varphi \vdash A \qquad \varphi \vdash B}{\varphi \vdash A \wedge B}$$

   From the I.H follows that $\varphi \models A$ and $\varphi \models B$. From the definition of $\models$ we have that $\varphi \models A \wedge B$ if $\varphi \models A$ and $\varphi \models B$, so we're done.

3. Case (funRight)

$$\frac{\varphi \vdash t_1 \qquad \dots \qquad \varphi \vdash t_n}{\varphi \vdash f(t_1,\dots,t_n)}$$

From the I.H follows that $\varphi \models t_1$ through $t_n$. Since $\models$ is closed by function application, then $\varphi \models f(t_1,\dots,t_n)$.

4. Case (AttLeft)

$$\frac{\Gamma, f(t_1,\dots,t_n) \vdash C}{\Gamma, t_1,\dots,t_n \vdash C}$$

From the I.H we have that $\Gamma, f(t_1,\dots,t_n) \models C$, but the set inclusion used in $\models$ is closed by function application, we can apply $f$ to $t_1,\dots,t_n$ and therefore $\Gamma, t_1,\dots,t_n \models C$.

5. Case (DestrLeft)

$$\frac{\Gamma, s \vdash C}{\Gamma, f(t_1,\dots,t_n) \vdash C}$$

From the I.H we have that $\Gamma, s \models C$. $\Gamma, f(t_1,\dots,t_n) \models C$ follows from the fact that $f(t_1,\dots,t_n)$ reduces to $s$ and the set inclusion used in $\models$ is closed by reduction, therefore $\Gamma, f(t_1,\dots,t_n) \models C$.

□

## A.5 Proof of Theorem 3.3.3

Given any finite term set $S$, the computation of $b(S)$ always reaches a saturation point and hence always terminates.

*Proof:* We define a measure on the sets generated by $\Rightarrow$ and show that this measure decreases monotonically in the sequence $S \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$.

Given a signature $\Sigma$, a base set $S$ and the maximum depth $K$ which is the sum of the maximum nesting depth for all available rewrite rules with the maximum nesting depth of terms in $S$, we define the cardinality $L$ of the set that contains all terms with maximum nesting smaller than $K$ as:

$$L = \#\{t \in T(\Sigma, S) \mid |t| < K\}$$

where $T(\Sigma, S)$ is the set of all terms generated from signature $\Sigma$ using the names present in $S$. We can now define the measure $|S_i|$ as: $|S_i| = L - \#S_i$

Considering the sequence: $S_0 \Rightarrow \cdots \Rightarrow S_n$ we can prove that for any $S_i$, $|S_{i+1}| < |S_i|$ by looking at the way $S_{i+1}$ is built from $S_i$.

In each step of $\Rightarrow$, a new term $M$ is inserted in the previous set. Also, by the constraints imposed by $\Rightarrow$, we know that $|M| < K$. Seeing as the rule for the application of $\Rightarrow$ is:

$$S, t_1, \ldots, t_n \Rightarrow S, t_1, \ldots, t_n, M$$

$$\text{if } \delta(g_1^{a_1}[t_1, \ldots, t_k], \ldots, g_i^{a_k}[t_{k'}, \ldots, t_n]) \rightarrow M, \forall i : a_i < K$$

and considering we impose that the right side of a reduction is a subterm of the left, and $\forall i : a_i < K$, we are sure that $|M| < K$.

In fact, the size $L$ is the maximum possible size for any set $S_i$, since $L$ is the size of the largest possible set of terms built from functions of $\Sigma$, whose nesting depth is smaller or equal to the maximum allowed in the rewrite rules for the functions in $\Sigma$.

Looking at $|S_i|$, we know the measure is strictly positive, because the existence of $S_{i+1}$ implies that $\#S_i < L$, since $L$ is the size of the maximum possible set of terms, and $\#S_{i+1} > \#S_i$ (because $S_{i+1}$ is obtained from $S_i$ by adding one new term $M$).

From the previous argument, it clearly follows that $|S_{i+1}| < |S_i|$, since $|S_i| > 0$, $|S_{i+1}| \geq 0$, $L$ is fixed and $\#S_{i+1} > \#S_i$. Noting that it is only possible for $|S_{i+1}| = 0$ if $S_{i+1} = S_n$, by the definition of $L$.

Given that $|S_0| > |S_1| > \cdots > |S_n|$, the approximation procedure is proven to terminate. □

## A.6 Proof of Theorem 3.3.4

Given a set of terms $S$, we have that:

$$\forall M \in \mathfrak{F}(S), \exists\, C, \bar{t} \in b(S) : M = C[\bar{t}]$$

Where $C$ is an arbitrary function application context.
*Proof:* Induction on the generation of $M \in \mathfrak{F}^{n+1}(S)$. We use $\mathfrak{F}^n(S)$ to denote the nth iteration over the saturation.

1. Case: $M \in S$

   Trivially satisfied by the empty context since for every $M \in S$, $M \in b(S)$.

2. Case: $M = f(M_1, \ldots, M_n)$ such that $\forall i : M_i \in \mathfrak{F}^n(S)$

   From the I.H we have that $\exists C_i, \bar{t_i} \in b(S) : M_i = C_i[\bar{t_i}]$ and thus we have the context $C = f(C_1[-], \ldots, C_n[-])$ such that $M = C[\bar{t_1}, \ldots, \bar{t_n}]$

3. Case: $\delta(M_1, \ldots, M_n) \rightarrow M$ such that $\forall i : M_i \in \mathfrak{F}(S)$

   From the I.H we have that $\exists C_i, \bar{t_i} \in b(S) : M_i = C_i[\bar{t_i}]$. Since $\delta(M_1, \ldots, M_n) \rightarrow M$ and we impose that the right-side of the rewrite rule to be a subterm of the left-side, we know that $M$ is a subterm of some $C_i[\bar{t_i}]$. Therefore, the depth of any term $M$ has to be such that:

   $$C'[\bar{t'}] = M$$

where $C'$ is a function context with nesting depth smaller than any $C_i$, and $\bar{t}'$ is a subset of all the $t_i$. This is so due to $M$ being such that it can be constructed by a context sub-composed of the several contexts $C_i$ that make up the left-side of the rewrite rule.

□

## A.7 Proof of Theorem 3.3.5

Given a set of terms $S$, if $M \in b(S)$ then $S \vdash M$.
*Proof:* Let $M \in b(S)$. Then there is a sequence $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$, where $M \in S_n$. We proceed by induction on $n$.

- Case $n = 0$:

$$\dfrac{}{S \vdash M}\ \text{(Id)}$$

- Case $n + 1$:

  From $\Rightarrow$ we have that $S_{n+1}$ is generated through:

$$R, t_1, \dots, t_n \Rightarrow R, t_1, \dots, t_n, M \;\text{ with } S_n = R, \bar{t} \text{ and } S_{n+1} = R, \bar{t}, M$$

$$\text{if } \delta(g_1^{a_1}[t_1, \dots, t_k], \dots, g_i^{a_k}[t_{k'}, \dots, t_n]) \rightarrow M, \forall i : a_i < K$$

  therefore, from the I.H we have that $S \vdash t_1, \dots, S \vdash t_n$. We can therefore build a derivation:

$$\dfrac{\dfrac{\vdots}{S \vdash \bar{t}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{S, \bar{t}, M \vdash M}\ \text{(Id)}}{S, \bar{t}, g_1^{a_1}[t_1, \dots, t_k], \dots, g_i^{a_k}[t_{k'}, \dots, t_n] \vdash M}\ \text{(DestrLeft)}}{\vdots}\ \text{(AttLeft)}}{S, \bar{t} \vdash M}\ \text{(AttLeft)}}{S \vdash M}\ \text{(Cut)}$$

  The schema for the derivation is a Cut, where the left premise follows from the I.H and the right premiss consists of a succession of applications of the (AttLeft) rule, which has the ability to apply arbitrary function symbols (of adequate arity) to terms on the left-side. With these applications, we reach a point where we have a term context that can match with the adequate rewrite rule that reduces to $M$. We then apply the (DestrLeft) rule to perform the reduction and conclude by using (Id). □

# Bibliography

[1] Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. *Theor. Comput. Sci.*, 367(1):2–32, 2006.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115, New York, NY, USA, 2001. ACM.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, New York, NY, USA, 1997. ACM.

[4] Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptol.*, 20(3):395–395, 2007.

[5] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[6] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM.

[7] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[8] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Flow logic for dolev-yao secrecy in cryptographic processes, 2002.

[9] Michele Boreale and Maria Grazia Buscemi. A framework for the analysis of security protocols. In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, pages 483–498, London, UK, 2002. Springer-Verlag.

[10] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[11] Luis Caires. Behavioral and spatial observations in a logic for the pi-calculus. *FoSSaCS 2004*, 2004.

[12] Luis Caires. Dynamical spatial logics: A tutorial survey. In *Bulletin of the EATCS*. 2008.

[13] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part ii). In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, pages 209–225, London, UK, 2002. Springer-Verlag.

[14] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2):194–235, 2003.

[15] Luís Caires and Etienne Lozes. Elimination of quantifiers and undecidability in spatial logics for concurrency. *Theor. Comput. Sci.*, 358(2):293–314, 2006.

[16] Luis Caires and Luis Monteiro. Verifiable and executable logic specifications of concurrent objects in lpi. In *ETAPS/ESOP98 Proceedings*. 1998.

[17] Luís Caires and Hugo Torres Vieira. Extensionality of spatial observations in distributed systems. *Electron. Notes Theor. Comput. Sci.*, 175(3):131–149, 2007.

[18] Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. In *ACM Workshop on Types in Language Design and Implementation, TLDI'03*, 2003.

[19] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 2000. ACM.

[20] Rohit Chadha, Stéphanie Delaune, and Steve Kremer. Epistemic logic for the applied pi calculus. In David Lee, Antonia Lopes, and Arnd Poetzsch-Heffter, editors, *Proceedings of IFIP International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE'09)*, Lecture Notes in Computer Science, Lisbon, Portugal, June 2009. Springer.

[21] Witold Charatonik and Jean-Marc Talbot. The decidability of model checking mobile ambients. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 339–354, London, UK, 2001. Springer-Verlag.

[22] Mika Cohen and Mads Dam. A complete axiomatization of knowledge and cryptography. *Logic in Computer Science, Symposium on*, 0:77–88, 2007.

[23] Mads Dam. Model checking mobile processes (full version). Technical report, 1993.

[24] Francien Dechesne, MohammadReza Mousavi, and Simona Orzan. Operational and epistemic approaches to protocol analysis: Bridging the gap. In *Proceedings of the 14th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'07)*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 226–241. Springer-Verlag, 2007.

[25] Richard A. DeMillo, Nancy A. Lynch, and Michael Y. Merritt. Cryptographic protocols, 1982.

[26] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, 1981.

[27] T. Dierks and C. Allen. The tls protocol version 1.0, 1999.

[28] Whitfield Diffie and Martin E. Hellman. New directions in cryptography, 1976.

[29] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Stanford, CA, USA, 1981.

[30] Étienne Lozes and Jules Villard. A spatial equational logic for the applied Π-calculus. In *CONCUR '08: Proceedings of the 19th international conference on Concurrency Theory*, pages 387–401, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Alan Freier, Philip Karlton, and Paul C. Kocher. The ssl protocol version 3.0, 1996.

[32] Gerhard Gentzen. Investigations into logical deduction. In *The Collected Works of Gerhard Gentzen*. North-Holland Publishing Company, 1934.

[33] Giorgio Ghelli and Giovanni Conforti. Decidability of freshness, undecidability of revelation. In *In Proc. of International Conference on Foundations of Software Science and Computational Structures (FOSSACS), volume 2987 of LNCS*, pages 105–120. Springer-Verlag, 2004.

[34] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[35] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[36] Daniel Hirschkoff. An extensional spatial logic for mobile processes. In *Proceedings of CONCUR'02*, 2002.

[37] American National Standards Institute. Triple data encryption algorithm modes of operation. Technical report, ANSI, 1998.

[38] F. Javier, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct, 1999.

[39] J. Kohl and C. Neuman. The kerberos network authentication service (v5), 1993.

[40] Dexter Kozen. Results on the propositional $\mu$-calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 348–359, London, UK, 1982. Springer-Verlag.

[41] Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *In Proc. 14th European Symposium On Programming (ESOP'05), volume 3444 of LNCS*, pages 186–200. Springer, 2005.

[42] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[43] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[44] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[45] Robin Milner. Functions as processes. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 167–180, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[46] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. In *CONCUR '91: Proceedings of the 2nd International Conference on Concurrency Theory*, pages 45–60, London, UK, 1991. Springer-Verlag.

[47] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[48] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, 1992.

[49] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[50] National Bureau of Standards. Data encryption standard. Technical report, National Bureau of Standards, 1977.

[51] National Institute of Standards and Technology. Advanced encryption standard. Technical report, NIST, 2001.

[52] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5, 1989.

[53] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *Theor. Comput. Sci.*, 299(1-3):451–475, 2003.

[54] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *TAPSOFT '93: Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 151–166, London, UK, 1993. Springer-Verlag.

[55] Davide Sangiorgi. A theory of bisimulation for the $\pi$-calculus. *Acta Inf.*, 33(1):69–97, 1996.

[56] Steve Schneider. Verifying authentication protocols with csp. In *In Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 3–17. IEEE Computer Society Press, 1997.

[57] Steve Schneider. Formal analysis of a non-repudiation protocol. pages 54–65. IEEE Computer Society Press, 1998.

[58] Douglas Stinson. *Cryptography: Theory and Practice*. Discrete Mathematics and its Applications. Chapman and Hall / CRC, 1995.

[59] Björn Victor and Faron Moller. The mobility workbench - a tool for the pi-calculus. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 428–440, London, UK, 1994. Springer-Verlag.

[60] Hugo Vieira, Luís Caires, and Ruben Viegas. The spatial logic model checker user's manual v1.0. Technical report, DI/FCT/UNL, 2005.

[61] T. Ylonen and C. Lonvick. The secure shell (ssh) authentication protocol, 2006.

[62] Jianying Zhou and Dieter Gollmann. A fair non-repudiation protocol. pages 55–61. IEEE Computer Society Press, 1995.