BSP Algorithms for LTL & CTL* Model Checking of Security Protocols

Michael Guedj

Université Paris-Est École Doctorale SIMME

Numéro attribué par la bibliothèque : .....

Thèse

pour obtenir le grade de
Docteur de l'université de Paris-Est
discipline : Informatique
présentée et soutenue publiquement par

Michael Guedj

le 11 octobre 2012

BSP Algorithms for LTL & CTL* Model Checking of Security Protocols

Composition du jury

Président :	Pr. Catalin Dima	Univ. of Paris-East
Rapporteurs :	Pr. Frédéric Loulergue	Univ. of Orléans
	Pr. Jean-François Prada-Peyre	Univ. of Paris-West
Examinateur :	Pr. Laure Petrucci	Univ. of Paris-North
	Pr. Hanna Klaudel	Univ. of Évry
Directeurs Scientifiques :	Dr. Frédéric Gava	Univ. of Paris-East
	Pr. Franck Pommereau	Univ. of Évry
Directeur :	Pr. Gaétan Hains	Univ. of Paris-East

Acknowledgements

The research thesis was done under the supervision of A/Prof Frédéric Gava, Prof. Franck Pommereau, and Prof. Gaetan Hains, in the University Paris-Est Creteil – UPEC.

The generous financial help of Digiteo and the Ile-de-france region is gratefully acknowledged.

To my family.

To Frédéric and Franck, for been the most dedicated supervisors anyone could hope for. For holding my hand and standing behind me all the way through. I thank you from my bottom of my heart.

My thanks go also to all the members of our Algorithmic, Complexity and Logic Laboratory (LACL) Laboratory.

Contents

1 Introduction **1**
- 1.1 Security Protocols . . . 2
  - 1.1.1 Generalities . . . 2
  - 1.1.2 Example . . . 3
  - 1.1.3 Motivations . . . 4
  - 1.1.4 Informal definition of security protocols . . . 4
  - 1.1.5 Security Properties and possible “attacks” . . . 5
  - 1.1.6 Why cryptographic protocols go wrong? . . . 8
- 1.2 Modelisation . . . 8
  - 1.2.1 High-level Petri nets . . . 8
  - 1.2.2 A syntactical layer for Petri nets with control flow: ABCD . . . 13
- 1.3 Parallelisation . . . 18
  - 1.3.1 What is parallelism? . . . 18
  - 1.3.2 Bulk-Synchronous Parallelism . . . 21
  - 1.3.3 Other models of parallel computation . . . 23
- 1.4 Verifying security protocols . . . 26
  - 1.4.1 Verifying security protocols through theorem proving . . . 26
  - 1.4.2 Verifying security protocols by model checking . . . 27
  - 1.4.3 Dedicated tools . . . 28
- 1.5 Model checking . . . 30
  - 1.5.1 Local (on-the-fly) and global model-checking . . . 30
  - 1.5.2 Temporal logics . . . 30
  - 1.5.3 Reduction techniques . . . 30
  - 1.5.4 Distributed state space generation . . . 35
- 1.6 Outline . . . 38

2 Stace space **41**
- 2.1 Security protocols as Label Transition System . . . 42
  - 2.1.1 Label Transition System and the marking (state) graph . . . 42
  - 2.1.2 LTS representation of security protocols . . . 42
  - 2.1.3 From LTS to high-level Petri nets . . . 42
  - 2.1.4 Sequential state space algorithm . . . 44
- 2.2 A naive parallel algorithm . . . 44
- 2.3 Dedicated parallel algorithms . . . 46
  - 2.3.1 Our generic protocols model . . . 46
  - 2.3.2 Having those structural informations from ABCD models . . . 47
  - 2.3.3 Increasing local computation time . . . 47
  - 2.3.4 Decreasing local storage: sweep-line reduction . . . 49
  - 2.3.5 Balancing the computations . . . 49
- 2.4 Formal explanations of the LTS hypothesis . . . 51
  - 2.4.1 General assumptions . . . 51
  - 2.4.2 Slices . . . 53

2.4.3 Receptions and classes . . . 54
2.4.4 Termination of the algorithms . . . 55
2.4.5 Balance considerations . . . 55

3 Model checking **57**
3.1 Tarjan . . . 57
3.1.1 Recursive Tarjan algorithm . . . 58
3.1.2 Iterative Tarjan algorithm . . . 59
3.2 Temporal logics LTL and CTL* . . . 60
3.2.1 Notations . . . 61
3.2.2 CTL* syntax and semantics . . . 61
3.2.3 Proof-structures for verifying a LTL formula . . . 63
3.3 LTL checking . . . 65
3.3.1 Sequential recursive algorithm for LTL . . . 65
3.3.2 Sequential iterative algorithm for LTL . . . 67
3.3.3 Parallel algorithm for LTL . . . 68
3.4 CTL* checking . . . 70
3.4.1 Sequential algorithms for CTL* . . . 73
3.4.2 Naive parallel algorithm for CTL* . . . 78
3.4.3 Parallel algorithm for CTL* . . . 85

4 Case study **95**
4.1 Specification of some security protocols using ABCD . . . 95
4.1.1 Modelisation of the security protocols . . . 95
4.1.2 Full example: the Needham-Schroeder protocol . . . 99
4.1.3 Other examples of protocols . . . 102
4.2 Implementation of the algorithms . . . 105
4.2.1 BSP programming in Python . . . 105
4.2.2 SNAKES toolkit and syntactic layers . . . 110
4.2.3 Parallel algorithms . . . 113
4.3 State space generation's benchmarks . . . 117
4.4 LTL and CTL*'s benchmarks . . . 119

5 Conclusion **131**
5.1 Summary of contributions . . . 132
5.2 Future works . . . 133

Bibliography **135**

1 Introduction

Contents

1.1 Security Protocols 2
1.1.1 Generalities 2
1.1.2 Example 3
1.1.3 Motivations 4
1.1.4 Informal definition of security protocols 4
1.1.5 Security Properties and possible “attacks” 5
1.1.6 Why cryptographic protocols go wrong? 8
1.2 Modelisation 8
1.2.1 High-level Petri nets 8
1.2.2 A syntactical layer for Petri nets with control flow: ABCD 13
1.3 Parallelisation 18
1.3.1 What is parallelism? 18
1.3.2 Bulk-Synchronous Parallelism 21
1.3.3 Other models of parallel computation 23
1.4 Verifying security protocols 26
1.4.1 Verifying security protocols through theorem proving 26
1.4.2 Verifying security protocols by model checking 27
1.4.3 Dedicated tools 28
1.5 Model checking 30
1.5.1 Local (on-the-fly) and global model-checking 30
1.5.2 Temporal logics 30
1.5.3 Reduction techniques 30
1.5.4 Distributed state space generation 35
1.6 Outline 38

In a world strongly dependent on distributed data communication, the design of secure infrastructures is a crucial task. Distributed systems and networks are becoming increasingly important, as most of the services and opportunities that characterise the modern society are based on these technologies. Communication among agents over networks has therefore acquired a great deal of research interest. In order to provide effective and reliable means of communication, more and more communication protocols are invented, and for most of them, security is a significant goal.

It has long been a challenge to determine conclusively whether a given protocol is secure or not. The development of formal techniques that can check various security properties is an important tool to meet this challenge. This document contributes to the development of such techniques by model security protocols using an algebra of coloured Petri net call ABCD and reduce time to checked the protocols using parallel computations. This allow parallel machines to apply automated reasoning techniques for performing a formal analysis of security protocols.

1.1 Security Protocols

1.1.1 Generalities

Cryptographic protocols are communication protocols that use cryptography to achieve security goals such as secrecy, authentication, and agreement in the presence of adversaries.

Designing security protocols is complex and often error prone: various attacks are reported in the literature to protocols thought to be "correct" for many years. These attacks exploit weaknesses in the protocol that are due to the complex and unexpected interleavings of different protocol sessions as well as to the possible interference of malicious participants.

Furthermore, they are not as easy that they appear [20]: the attacker is powerful enough to perform a number of potentially dangerous actions as intercepting messages flowing over the network, or replacing them by new ones using the knowledge he has previously gained; or is able to perform encryption and decryption using the keys within his knowledge [77]. Consequently the number of possible testing attacks generally growing exponentially of the size of the session.

Formal methods offer a promising approach for automated security analysis of protocols: the intuitive notions are translated into formal specifications, which is essential for a careful design and analysis, and protocol executions can be simulated, making it easier to verify certain security properties. Formally verifying security protocols is now an old subject but still relevant. Different approach exist as [10, 12, 101] and tools were dedicated for this work as [11, 68].

To establish the correctness of a security protocol, we first need to define a model in which such protocol is going to be analyzed. An analysis model consists of three submodels: a property model, an attacker model, and an environment model. The environment model encloses the attacker model, while the property model is separate. The property model allows the formalization of the goals of the protocol, that is the security guarantees it is supposed to provide. The security goals are also known as the protocol requirements or the security properties. The attacker model describes a participant, called the attacker (or intruder) which does not necessarily follow the rules of the protocol. Actually, its main interest is in breaking the protocol, by subverting the intended goal — specified using the property model described above. In the attacker model, we detail which abilities are available to the attacker, that is, which operations the attacker is able to perform when trying to accomplish its goal. The attacker model is also sometimes called the threat model.

The environment model is a representation of all the surrounding world of the attacker — described above in the attacker model. The environment model includes honest principals which faithfully follow the steps prescribed by the security protocol. By modelling these principals, the environment model also encodes the security protocol under consideration. Furthermore, the environment model describes the communication mechanisms available between participants. Alternatively, the environment model may also describe any quality of interest from the real world which may influence the behaviour (or security assurances) of the protocol. Examples include modelling explicitly the passage of time, or modelling intrinsic network characteristics such as noise or routing details.

The attacker is assumed to have complete network control. Thus, the attacker can intercept, block or redirect any communication action executed by an honest principal. The attacker can also synthesize new messages from the knowledge it has, and communicate these messages to honest participants. This synthesis, which is the ability to create new messages, is precisely defined. However, if the attacker does not know the correct decryption key of a given ciphertext (*i.e.* an encrypted message), then it can not gain any information from the ciphertext. This assumption is crucial, and it is known as perfect or ideal encryption. Hence, the attacker is not assumed to be able to cryptanalyse the underlying encryption scheme, but simply treat it as perfect. As with encryption, every cryptographic primitive available to the attacker (*e.g.* hashing or signature) is similarly idealized, arriving at perfect cryptography.

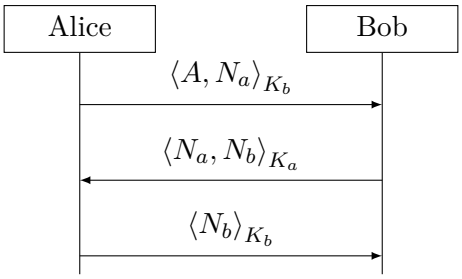


Figure 1.1. An informal specification of NS protocol, where N_a and N_b are nonces and K_a, K_b are the public keys of respectively Alice and Bob.

1.1.2 Example

The protocol NS involves two agents Alice (A) and Bob (B) who want to mutually authenticate. This is performed through the exchange of three messages as illustrated in Figure 1.1. In this specification, a message m is denoted by $\langle m \rangle$ and a message encrypted by a key k is denoted by $\langle m \rangle_k$ — we use the same notation for secret key and public key encryption. The three steps of the protocol can be understood as follows:

1. Alice sends her identity A to Bob, together with a nonce N_a; The message is encrypted with Bob's public key K_b so that only Bob can read it; N_a thus constitutes a challenge that allows Bob to prove his identity; Bob is the only one who can read the nonce and send it back to Alice;

2. Bob solves the challenge by sending N_a to Alice, together with another nonce N_b that is a new challenge to authenticate Alice;

3. Alice solves Bob's challenge, which results in mutual authentication.

This protocol is well known for being flawed when initiated with a malicious third party Mallory (M). Let us consider the run depicted in Figure 1.2. It involves two parallel sessions, with Mallory participating in both of them:

- when Mallory receives Alice's first message, she decrypts it and forwards to Bob the same message (but encrypted with Bob's key) thus impersonating Alice;
- Bob has no way to know that this message is from Mallory instead of Alice, so he answers exactly as in the previous run;
- Mallory cannot decrypt this message because it is encrypted with Alice's key, but she might use Alice has an oracle and forward the message to her
- when Alice receives $\langle N_a, N_b \rangle_{K_a}$, she cannot know that this message has been generated by Bob instead of Mallory, and so she believes that this is Mallory's answer to her first message;
- so, Alice sends the last message of her session with Mallory who is now able to retrieve N_b and authenticate with Bob.

In this attack, both sessions (on the left and on the right) are perfectly valid according to the specification of the protocol. The flaw is thus really in the protocol itself, which is called a *logical attack*. This can be easily fixed by adding the identity of the sender to each message (like in the first one), in which case Alice can detect that the message forwarded by Mallory (now it is $\langle B, N_a, N_b \rangle_{K_a}$) is originated from Bob.

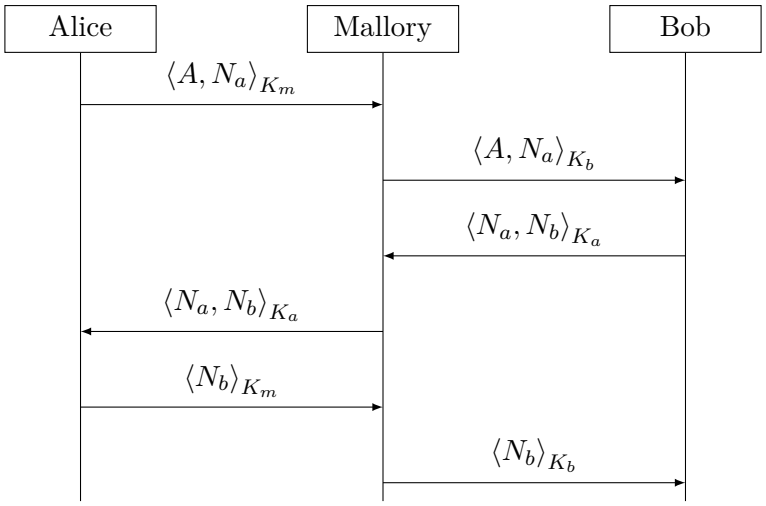


Figure 1.2. An attack on NS protocol where Mallory authenticates as Alice with Bob.

1.1.3 Motivations

The possibility of violations and attacks of security protocols sometimes stems from subtle misconceptions in the design of the protocols. Typically, these attacks are simply overlooked, as it is difficult for humans, even by careful inspection of simple protocols, to determine all the complex ways that different protocol sessions could be interleaved together, with possible interference of a malicious intruder, the attacker.

The question of whether a protocol indeed achieves its security requirements or not is, in the general case, undecidable [4,82,90]. This has been proved by showing that a well-known undecidable problem (*e.g.* the Post Correspondence Problem, the halting problem for Turing machines, *etc.*) can be reduced to a protocol insecurity problem. Despite this strong undecidability result, the problem of deciding whether a protocol is correct or not it is still worthwhile to be tackled by the introduction of some restrictions can lead to identify decidable subclasses: by focusing on verification of a bounded number of sessions the problem is known to be NP-complete. This can be done by simply enumerating and exploring all traces of the protocol's state transition system looking for a violation to some of the requirements. However the specific nature of security protocols that make them particularly suited to be checked by specific tools. That also need how formalise those protocols to be latter checked.

Although, if the general verification problem is undecidable, for many protocols, verification can be reduced to verification of a bounded number of sessions. Moreover, even for those protocols that should theorically be checked under a unbounded number of concurrent protocol executions, violations in their security requirements often exploit only a small number of sessions. For these reasons, in many cases of interest it is sufficient to consider a finite number of sessions in which each agent performs a fixed number of steps. For instance all the attacks on the well-know SPORE and Clark-Jacob's libraries [51] can be discovered by modelling each protocol with only two protocol sessions.

1.1.4 Informal definition of security protocols

Communication protocols specify an exchange of messages between principals, *i.e.* the agents participating in a protocol execution — *e.g.* users, hosts, or processes. Messages are sent over open networks, such as the Internet, that cannot be considered secure. As a consequence, protocols should be designed "robust" enough to work even under worst-case assumptions, namely messages may be eavesdropped or tampered with by an intruder or dishonest or careless princi-

pals. A specific category of protocols has been devised with the purpose of securing communications over insecure networks: security (or cryptographic) protocols are communication protocols that aim at providing security guarantees such as authentication of principals or secrecy of some piece of information through the application of cryptographic primitives.

The goal of cryptographic is to convert a plain-text P into a cipher-text C (and *vice versa*) that is unintelligible to anyone (a spy) that monitoring the network. The process of converting P into C is called encryption, while the reverse procedure is called decryption. The main feature of computer's encryption is the used of an additional parameter K known as the encryption key. In order to recover the original plain-text the intended receiver should use a second key K^{-1} called the inverse key where is no way to compute easally it from K — and *vice versa*.

The best-known cryptographic algorithms for key are the well-known DES (Digital Encryption Standard) and the RSA (Rivest, Shamir, and Adleman) algorithms. The security of cryptographic algorithms relies in the difficulty of breaking them by performing a brute-force search of the key space. Hence the use of keys sufficiently long to prevent a brute force attack in a reasonable time entirely justifies the standard assumption adopted in formal analysis of security protocols and called perfect cryptography. The idea underlying such an assumption is that an encrypted message can be decrypted only by using the appropriate decryption key, *i.e.* it is possible to retrieve M from M_K only by using K^{-1} as decryption key.

Protocols are normally expressed as narrations, where some finer details are abstracted away. A protocol narration is a simple sequence of message exchanges between the different participating principals and can be interpreted as the intended trace of the ideal execution of the protocols. Informally, the scenario we are interesting in involves a set of honest agents that, according to a security protocol, exchange messages over insecure communication channels controlled by a malicious agent called intruder with the ultimate goal of achieving some security requirements. Participants (agents) perform sequence of data exchange (sending or received operators) which could be seen as "ping-pong".

1.1.5 Security Properties and possible "attacks"

What kind of attacks do there exist against security properties of protocol ? This question cannot be answered before having defined what we expect from a given security protocol. We give here an informal definition of possible and well-known "attacks" and security properties as well as some vocabulary of protocols.

Vocabulary. Let us recall some elemetary vocabulary on security protocols:

- **Fresh Terms**. A protocol insecurity problem can allow for the generation of fresh terms *e.g.* nonce this allow to have a new value each time the protocol is used; random numbers from the system can be used;
- **Step**. The number of steps that an honest agent can perform to execute a session of the protocol;
- **Sessions**. An agent can execute more than one time the protocol; each use of the protocol is call a session;
- **Agents**. The participants of the protocols including intruders.

In general, the cryptographic protocol consists of agents who are willing to engage in a secure communication using an insecure network and sometime using trusted server, which generates the fresh session key used for exchanging data securely between the principals. The session key is abandoned after data exchanging is over. In fact, it is not possible to establish an authenticated session key without existing secure channels already being available [38]. Therefore it is essential that some keys are already shared between different principals, which are often referred to as master keys. Different from session keys, which expire after each session, master keys are changed

less frequently, and consequently leaking master keys always causes cryptographic protocols to be broken.

Fail in security protocols. The history of security protocols is full of examples, where weaknesses of supposedly correct published protocols that have been implemented and deployed in real applications only to be found flawed years later. The most well-known case is the Needham-Schroeder authentication protocol that was found vulnerable to a man-in-the-middle attack 17 years after its publication. It has been shwo by "The Computer Security Institute"[1] that the number of vulnerabilities of protocols is highly growing up and a discovering one of them is a daily thing for compagnies and researchers. But, generally speaking, security problems are undecidable for their dynamic behaviour due to, say, mis-behaved agents and unbounded sessions of protocol executions. Therefore, verification of security properties is an important research problem. This leads to the researches in searching for a way to verify whether a system is secure or not.

Security Attacks. Let us now enumerate some typical attacks. They can be categorised into the following:

- **Interruption**. The communications are destroyed or becomes unavailable or unusable; examples include destruction of a piece of hardware, *i.e.* a hard disk, or the cutting of a physical communication line, *i.e.* a cable. An agent (as a server or else) is then unattainable;
- **Eavesdropping**. An unauthorised party gains access to the communication; the unauthorised party could be a person, a program, or a computer; examples include wiretapping to capture data in a network, and the illegally copying of files or programs;
- **Modification**. An unauthorised party not only gains access to but tampers with the network; examples include changing values in a data file, altering a program so that it performs differently, and modifying the content of messages being transmitted in a network;
- **Fabrication**. An unauthorised party inserts counterfeit data into the network; examples include the inserting of spurious message in a network or the addition of records to a file;
- **Traffic analysis**. An unauthorised party intercepts and examines the messages flowing over the network in order to deduce information from the message patterns; it can be performed even when the messages are encrypted and can not be decrypted.

There are many kinds of attacking security protocol. Some well-known strategies that an intruder might employ are:

- **Man-in-the-middle**. This style of attack involves the intruder imposing himself between the communications between the sender and receiver; if the protocol is purely designed he may be able to subvert it in various ways; in particular he may be able to forge as receiver to sender, for example;
- **Replay**. The intruder monitors a run of the protocol and at some later time replays one or more of the messages; If the protocol does not have the mechanism to distinguish between separate runs or to detect the staleness of a message, it is possible to fool the honest agents into rerunning all or parts of the protocol; devices like nonces, identifiers for runs and timestamps are used to try to foil such attacks;
- **Interleave**. This is the most ingenious style of attack in which the intruder contrives for two or more runs of the protocol to overlap.

[1] http://www.gocsi.com

There are many other known styles of attack and presumably many more that have yet to be discovered. Many involve combinations of these themes. This demonstrates the difficulty in designing security protocols and emphasizes the need for a formal and rigorous analysis of these protocols.

A protocol execution is considered as involving honest (participants) principals and active attackers. The abilities of the attackers and relationship between participants and attackers together constitute a threat model and the almost exclusively used threat model is the one proposed by Dolev and Yao [77]. The Dolev-Yao threat model is a worst-case model in the sense that the network, over which the participants communicate, is thought as being totally controlled by an omnipotent attacker with all the capabilities listed above. Therefore, there is no need to assume the existence of multiple attackers, because they together do not have more abilities than the single omnipotent one. Dishonest principals do not need to be considered either: they can be viewed as an unique Dolev-Yao intruder. Furthermore, it is generally not interesting to consider an attacker with less abilities than the omnipotent one except to verify less properties and to accelerate the formal verification of a protocol.

Security properties. Each cryptographic protocol is designed to achieve one or more security-related goals after a successful execution, in other words, the principals involved may reason about certain properties; for example, only certain principals have access to particular secret information. They may then use this information to verify claims about subsequent communication, *e.g.* an encrypted message can only be decrypted by the principals who have access to the corresponding encryption key. The most commonly considered security properties include:

- **Authentication**. It is concerned with assuring that a communication is authentic; in the case of an ongoing interaction, such as the connection of a host to another host, two aspects are involved; first, at the time of connection initiation, the two entities have to be authentic, *i.e.* each is the entity that he claims to be; second, during the connection, there is no third party who interferes in such a way that he can masquerade as one of the two legitimate parties for the purposes of unauthorized transmission or reception; for example, fabrication is an attack on authenticity;

- **Confidentiality**. It is the protection of transmitted data from attacks; with respect to the release of message contents, several levels of protection can be identified, including the protection of a single message or even specificfields within a message; for example, interception is an attack on confidentiality;

- **Integrity**. Integrity assures that messages are received as sent, with no duplication, insertion, modification, reordering, or replays; as with confidenfitiality, integrity can apply to a stream of messages, a single message, or selected fields within a message; modification is an attack on integrity;

- **Availability**. Availability assures that a service or a piece of information is accessible to legitimate users or receivers upon request; there are two common ways to specify availability; an approach is to specify failure factors (factors that could cause the system or the communication to fail) [185], for example, the minimum number of host failures needed to bring down the system or the communication; interruption is, for example, an attack on availability;

- **Non-repudiation**. Non-repudiation prevents either sender or receiver from denying a transmitted message; thus, when a message is sent, the receiver can prove that the message was in fact sent by the alleged sender; similarly, when a message is received, the sender can prove that the message was in fact received by the alleged receiver.

1.1.6 Why cryptographic protocols go wrong?

The first reason for the security protocols easily go wrong is that protocols were first usually expressed as narrations and most of the details of the actual deployment are ignored. And this little details and ambiguities may be the reason of an attack.

Second, as mentioned before, cryptographic protocols are mainly deployed over an open network such that everyone can join it, exceptions are where wireless or routing protocols attacker control only a subpart of the network and where agents only communicate with their neighbors [13, 27, 121, 197, 198]. One reason for security protocols easily going wrong is the existence of the attacker: he can start sending and receiving messages to and from the principals across it without the need of authorization or permission. In such an open environment, we mush anticipate that the attacker will do all sorts of actions, not just passively eavesdropping, but also actively altering, forging, duplicating, re-directing, deleting or injecting messages. These fault messages can be malicious and cause a destructive effect to the protocol. Consequently, any message received from the network is treated to have been received from the attacker after his disposal. In other words, the attacker is considered to have the complete control of the entire network and could be considered to be the network. And it is easy for humans to forget a possible combination of the attacker. Instead, automatic verification (model-checking), which is the subject of this document, would not forget one possible attack. And this number of attack growing exponentially and reduce the time of computation of generating all these attacks using parallel machine is the main goal of this document.

It is notice to say that nowadays a considerable number of cryptographic protocols have been specified, implemented and verified. Consequently analysing cryptographic protocols in order to find various kinds of attacks and to prevent them has received a lot of attention. As mentioned before, the area is remarkably subtle and a very large portion of proposed protocols have been shown to be flawed a long time after they were published. This has naturally encouraged research in this area.

1.2 Modelisation

A more complete presentation is available at [181].

1.2.1 High-level Petri nets

(a) Definition of classical Petri nets

A Petri net (also known as a place/transition net or P/T net) is a simple model of distributed systems [175, 176]. A P/T net is a directed bipartite graph consists of places, transitions, and directed arcs.

Intuitivelly transitions represent events that may occur, directed arcs (also called edges) the data and control flow; and places are the ressources. Arcs run from a place to a transition (or *vice versa*, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition.

One thing that makes Petri nets interesting is that they provide a balance between modeling power and analyzability: it is "easy" to modelised many distributed system and many properties about the concurrency of the modelised system can be automatically determined. This is commonly called model-checking — we will give a better definition in the former.

It is standard to a use a graphical representation of the P/T nets. We used the following one:

- places are represented by circles;
- transitions are denoted by squares;
- arcs are denoted by arrows.

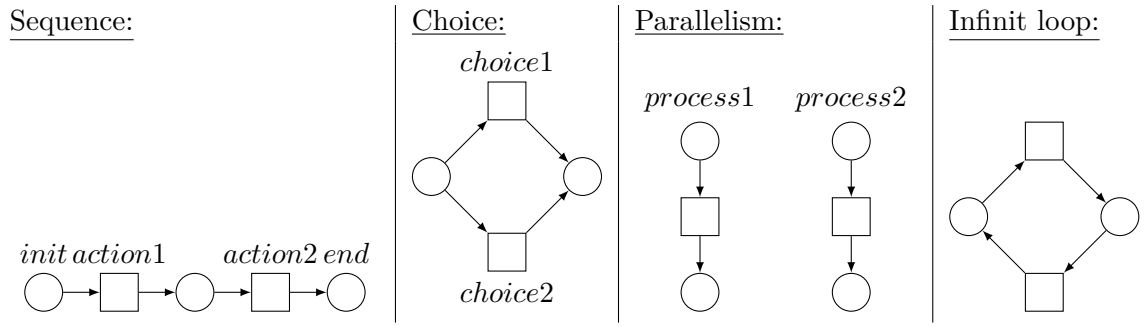


Figure 1.3. Petri Nets for sequence, choice, iteration and parallelism

It is common to give name to transitions and places for a better read of the net. As show in Figure 1.3, Petri nets can easally model classical structure of distributed system such as sequence, choice, iteration or parallelism. Now, we give here a formal definition of Petri nets:

Definition 1 (*Petri nets (P/T)*).
A Petri net *is a tuple* (S, T, ℓ) *where:*

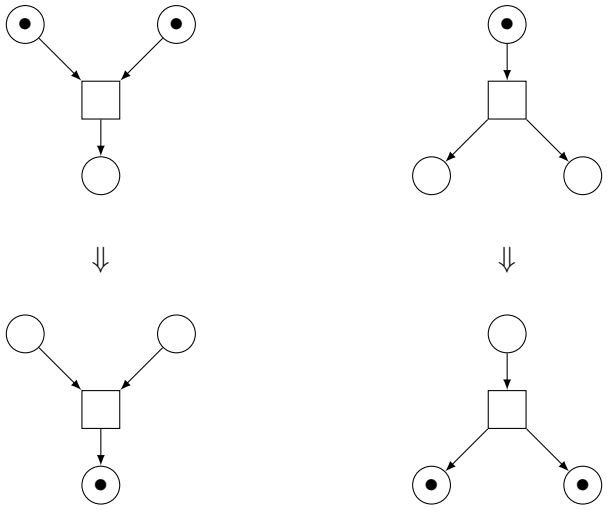
- S *is the finite set of* places*;*
- T*, disjoint from* S (i.e. $T \cap S = \emptyset$)*, is the finite set of* transitions*;*
- ℓ *is a* labelling *function such that for all* $(x, y) \in (S \times T) \cup (T \times S)$*,* $\ell(x, y)$ *is a multiset over* $\mathbb{E}$ *and defines the* arc *from* x *toward* y*.*

Each place can hold a number of individual *tokens* that represent data items flowing through the net. A transition is called enabled if there are tokens present at each of its input places, and if all output places have not reached their capacity. Enabled transitions can *fire* by consuming a specified number of tokens from each of the input places and putting a number of new token on each of the output places.

The number of tokens held at each place is specified by the *marking* of the net, which represents the current state of the net. Consecutive markings are obtained by firing transitions. Informally, starting from an initial marking, computing the marking graph of a Petri net consist to compute all the consecutive markings. This problem is known to be EXPSPACE-hard and thus decidable. Papers continue to be published on how to do it efficiently which is in certain manner also the goal of this thesis. It is common to had bullets into places to represent markings.

Formally, a marking m is represented as a vector $\mathbb{N}^{|S|}$ with element $m[p]$ denoting the number of tokens on place p in marking m. Marking m_0 is the initial marking, *i.e.* it contains the number of tokens on each place at the beginning. A new marking m' that is obtained from marking m by firing transition t, noted $m[t\rangle m'$ can be computed if m has enough tokens, *i.e.* for all $s \in S$, $\ell(s, t) \leq m(s)$. Then m' is defined for all $s \in S$ as $m'(s) \stackrel{\text{df}}{=} m(s) - \ell(s, t) + \ell(t, s)$.

For example, two markings of two different P/T nets, each one firing on transition which give two new markings:



If P/T nets are a simple and convenient model for study, it a main drawbacks: tokens do not carry any value, and are undistinguishable. For example, conditional branches can only be nondeterministic and cannot depend on the value of the data. Also, using the P/N model for distributed systems requires to use, for instance, one buffer to represent each possible value of a modelled variable, which is not readable for large data types and may become humanly intractable in complex cases.

Because such dependences are central requirements for many distributed systems, P/T nets are not sufficient to entirely capture complex behaviors. Therefore, we use a more expressive Petri Net variant the High-Level(Coloured) Petri Nets, which we define in the next section.

(b) High-Level or Coloured Petri nets

High-Level Petri Nets also called Coloured Petri Nets, have the same structure as P/T nets, but tokens are now distinguishable ("coloured"), *i.e.* they carry values. Therefore, transitions do not only take and put tokens upon firing, but they can be restricted in what colours of tokens they accept, and can transform input tokens into differently coloured output tokens. This allows to express transitions that transform tokens.

Before defining Coloured Petri Nets, we first introduce the notion of multisets, *i.e.* sets that can contain the same element several times.

A *multiset* m over a domain D is a function $m : D \to \mathbb{N}$ (natural numbers), where, for $d \in D$, $m(d)$ denotes the number of occurrences of d in the multiset m. The empty multiset is denoted by $\emptyset$ and is equivalent to the function $\emptyset \stackrel{\text{df}}{=} (\lambda x.0)$. We shall denote multisets like sets with repetitions, for instance $m_1 \stackrel{\text{df}}{=} \{1, 1, 2, 3\}$ is a multiset and so is $\{d + 1 \mid d \in m_1\}$. The latter, given in extension, is denoted by $\{2, 2, 3, 4\}$. A multiset m over D may be naturally extended to any domain $D' \supset D$ by defining $m(d) \stackrel{\text{df}}{=} 0$ for all $d \in D' \setminus D$. If m_1 and m_2 are two multisets over the same domain D, we define:

- **order**: $m_1 \leq m_2$ iff $m_1(d) \leq m_2(d)$ for all $d \in D$;
- **union**: $m_1 + m_2$ is the multiset over D defined by $(m_1 + m_2)(d) \stackrel{\text{df}}{=} m_1(d) + m_2(d)$ for all $d \in D$;
- **difference**: $m_1 - m_2$ is the multiset over D defined by $(m_1 - m_2)(d) \stackrel{\text{df}}{=} \max(0, m_1(d) - m_2(d))$ for all $d \in D$;
- **membership**: for $d \in D$, we denote by $d \in m_1$ the fact that $m_1(d) > 0$.

A coloured Petri net involves values, variables and expressions. These objects are defined by a *colour domain* that provides data values, variables, operators, a syntax for expressions, possibly typing rules, *etc.* For instance, one may use integer arithmetic or Boolean logic as colour domains. Usually, more elaborated colour domains are useful to ease modelling, in particular,

one may consider a functional programming language or the functional fragment (expressions) of an imperative programming language. In most of this document, we consider an abstract colour domain with the following elements:

- $\mathbb{D}$ is the set of *data* values; it may include in particular the Petri net *black token* $\bullet$, integer values, Boolean values True and False, and a special "undefined" value $\perp$;
- $\mathbb{V}$ is the set of *variables*, usually denoted as single letters x, y, ..., or as subscribed letters like x_1, y_k, ...;
- $\mathbb{E}$ is the set of *expressions*, involving values, variables and appropriate operators. Let $e \in \mathbb{E}$, we denote by $\mathsf{vars}(e)$ the set of variables from $\mathbb{V}$ involved in e. Moreover, variables or values may be considered as (simple) expressions, *i.e.* we assume that $\mathbb{D} \cup \mathbb{V} \subset \mathbb{E}$.

We make no assumption about the typing or syntactical correctness of values or expressions; instead, we assume that any expression can be evaluated, possibly to $\perp$ (undefined). More precisely, a *binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D}$. Let $e \in \mathbb{E}$ and β be a binding, we denote by $\beta(e)$ the evaluation of e under β; if the domain of β does not include $\mathsf{vars}(e)$ then $\beta(e) \stackrel{\text{df}}{=} \perp$. The application of a binding to evaluate an expression is naturally extended to sets and multisets of expressions.

For instance, if $\beta \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto 2\}$, we have $\beta(x + y) = 3$. With $\beta \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto \texttt{"2"}\}$, depending on the colour domain, we may have $\beta(x + y) = \perp$ (no coercion), or $\beta(x + y) = \texttt{"12"}$ (coercion of integer 1 to string 1), or $\beta(x + y) = 3$ (coercion of string 2 to integer 2), or even other values as defined by the concrete colour domain.

Two expressions $e_1, e_2 \in \mathbb{E}$ are *equivalent*, which is denoted by $e_1 \equiv e_2$, iff for all possible binding β we have $\beta(e_1) = \beta(e_2)$. For instance, $x+1$, $1+x$ and $2+x-1$ are pairwise equivalent expressions for the usual integer arithmetic.

Definition 2 (*Coloured Petri nets*).
A Petri net *is a tuple* (S, T, ℓ) *where:*

- S *is the finite set of* places*;*
- T*, disjoint from* S*, is the finite set of* transitions*;*
- ℓ *is a* labelling *function such that:*
  - *for all* $s \in S$*,* $\ell(s) \subseteq \mathbb{D}$ *is the* type *of* s*,* i.e. *the set of values that* s *is allowed to carry,*
  - *for all* $t \in T$*,* $\ell(t) \in \mathbb{E}$ *is the* guard *of* t*,* i.e. *a condition for its execution,*
  - *for all* $(x, y) \in (S \times T) \cup (T \times S)$*,* $\ell(x, y)$ *is a multiset over* $\mathbb{E}$ *and defines the* arc *from* x *toward* y*.*

As usual, Coloured Petri nets are depicted as graphs in which places are round nodes, transitions are square nodes, and arcs are directed edges. See figure 1.4 for a Petri net represented in both textual xand graphical notations. Empty arcs, *i.e.* arcs such that $\ell(x, y) = \emptyset$, are not depicted. Moreover, to alleviate pictures, we shall omit some annotations (see Figure 1.4): $\{\bullet\}$ for place types or arc annotations, curly brackets $\{\,\}$ around multisets of expressions on arcs, True guards, and node names that are not needed for explanations.

For any place or transition $x \in S \cup T$, we define ${}^\bullet x \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(y, x) \neq \emptyset\}$ and, similarly, $x^\bullet \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(x, y) \neq \emptyset\}$. For instance, considering the Petri net of figure 1.4, we have ${}^\bullet t = \{s_1\}$, $t^\bullet = \{s_1, s_2\}$, ${}^\bullet s_2 = \{t\}$ and $s_2{}^\bullet = \emptyset$. Finally, two Petri nets (S_1, T_1, ℓ_1) and (S_2, T_2, ℓ_2) are *disjoint* iff $S_1 \cap S_2 = T_1 \cap T_2 = \emptyset$.

Definition 3 (*Markings and sequential semantics*).
Let $N \stackrel{\text{df}}{=} (S, T, \ell)$ *be a Petri net.*

A marking M *of* N *is a function on* S *that maps each place* s *to a finite multiset over* $\ell(s)$ *representing the tokens held by* s*.*

A transition $t \in T$ *is* enabled *at a marking* M *and a binding* β*, which is denoted by* $M[t, \beta\rangle$*, iff the following conditions hold:*

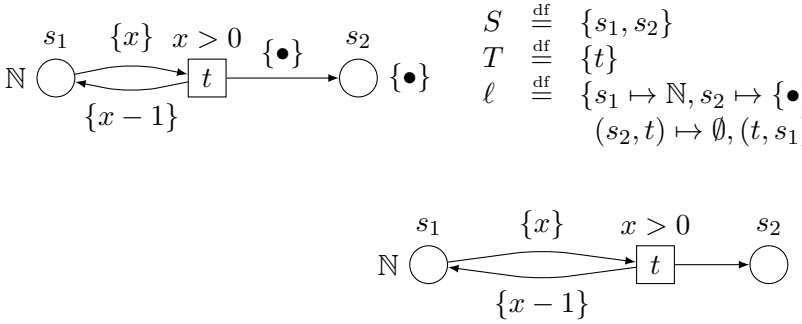


Figure 1.4. A simple Petri net, with both full (top) and simplified annotations (below).

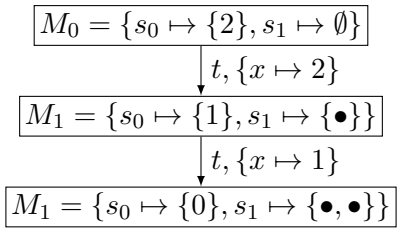


Figure 1.5. The marking graph of the Petri net of figure 1.4.

- *M has enough tokens,* i.e. *for all $s \in S$, $\beta(\ell(s,t)) \leq M(s)$;*
- *the guard is satisfied,* i.e. $\beta(\ell(t)) = \mathsf{True}$*;*
- *place types are respected,* i.e. *for all $s \in S$, $\beta(\ell(t,s))$ is a multiset over $\ell(s)$.*

If $t \in T$ is enabled at marking M and binding β, then t may fire *and yield a marking M' defined for all $s \in S$ as $M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$. This is denoted by $M[t,\beta\rangle M'$.*

The marking graph *G of a Petri net marked with M is the smallest labelled graph such that:*

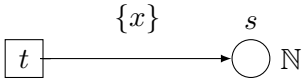
- *M is a node of G;*
- *if M' is a node of G and $M'[t,\beta\rangle M''$ then M'' is also an node of G and there is an arc in G from M' to M'' labelled by (t,β).*

Note that if $M' \in G$ then $M[t,\beta\rangle^* M'$ where $[t,\beta\rangle^*$ is the transitive and reflexive closure of $[t,\beta\rangle$.

It may be noted that this definition of marking graphs allows to add infinitely many arcs between two markings. Indeed, if $M[t,\beta\rangle$, there might exist infinitely many other enabling bindings that differ from β only on variables not involved in t. So, we consider only firings $M[t,\beta\rangle$ such that the domain of β is $\mathsf{vars}(t) \stackrel{\text{df}}{=} \mathsf{vars}(\ell(t)) \cup \bigcup_{s \in S}(\mathsf{vars}(\ell(s,t)) \cup \mathsf{vars}(\ell(t,s)))$.

For example, let us consider again the Petri net of figure 1.4 and assume it is marked by $M_0 \stackrel{\text{df}}{=} \{s_0 \mapsto \{2\}, s_2 \mapsto \emptyset\}$, its marking graph has three nodes as depicted in Figure 1.5. Notice that from marking M_2, no binding can enable t because, either $x \not\mapsto 0$ and then M_2 has not enough tokens, or $x \mapsto 0$ and then both the guard $x > 0$ is not satisfied and the type of s_1 is not respected — $x - 1$ evaluates to -1.

It may also be noted that the marking graph is not finite in general. Take for example:



would give a marking graph where each marking correspond to a natural. Making the graph finite is possible if for example all transitions and the colors domaines have a finite number of

inputs and ouputs. However, deciding if a marking graph is finite or not is the subject of this thesis. More detail can be found in [181].

It is notice that a simple solution (which is used in [181] and in this thesis) to the above problem is to forbid free variables. This is not an issue in practice since free variables usually result from either a mistake, or a need for generating a random value. Forbidding free variables prevents the corresponding mistakes and generating random values can be handled another way: add an input place containing all the values among which a random one has to be chosen; add a read arc or a self loop labelled by the variable that used to be free, from this place to the transition.

Noted also that restricting the colour domain is generally good for analysis capabilities and performances, but usually bad for ease of modelling. In the Petri Net's libraries used in this thesis (SNAKE see Section 4.2.2) it has been chosen to restrict annotations in a way which allowed to have no restriction on the colour domain — full Python language.

1.2.2 A syntactical layer for Petri nets with control flow: ABCD

To our purpose, that is security protocols, it is not convient to directly manipulating general Coloured Petri Net. In fact, we only need to manipulate sequential and deterministic processes (the agents of a protocol) that are fully in parallel and would communicate *via* the network or specific mediums.

The modelling of concurrent systems as security protocols involves a representation of inter-process communication. This representation should be compact and readable in order to avoid design errors. A first step for improving the readability is the structured and modular modelling which is a main characteristic of *box* process algebras. Boxes are like statements in a structured language (Boxes can also give a control flow of the processes) and users compose boxes to have the full model of the system. Processes as boxes are thus built on top of atomic actions and by recursive composition of boxes.

Considering our Petri nets as a semantics domain, it is thus possible to define languages adapted to specific usages, with a well defined semantics given in terms of Petri nets. In order to show how our framework makes this easy, we present now a syntax for Petri nets with control flow that embeds a programming language (which well be Python in this thesis) as colour domain. This language, called the *asynchronous box calculus with data* [179], or ABCD, is a syntax for Petri nets with control flow. ABCD is a specification language that allows its users to express the behavior concurrent systems at a high level. A main feature is that any ABCD expression would be translated into coloured Petri nets.

(a) Control flow operations

To define compositions of Petri nets as ABCD's exressions, we extend them with *node statuses*. Let $\mathbb{S}$ be the set of statuses, comprising: e, the status for *entry places*, *i.e.* those marked in an initial state of a Petri net; x, the status for *exit places*, *i.e.* those marked in a final state of a Petri net; i, the status for *internal places*, *i.e.* those marked in intermediary states of a Petri net; ε, the status of *anonymous places*, *i.e.* those with no distinguished status; arbitrary names, like *count* or *var*, for *named places*. Anonymous and named places together are called *data* or *buffer* places, whereas entry, internal and exit places together are called *control flow* places.

Definition 4 (*Petri nets with control flow*).
A Petri net with control flow *is a tuple* (S, T, ℓ, σ) *where:*

- (S, T, ℓ) *is a Petri net;*
- σ *is a function* $S \to \mathbb{S}$ *that provides a status for each place;*
- *every place* $s \in S$ *with* $\sigma(s) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$ *is such that* $\ell(s) = \{\bullet\}$.

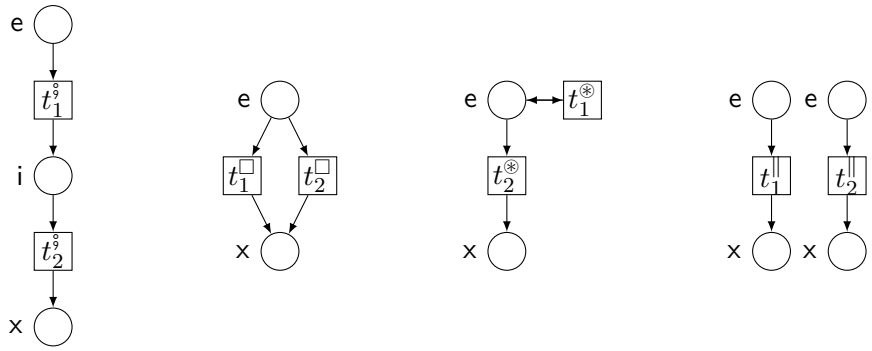


Figure 1.6. Operators nets.

Statuses are depicted as labels, except for ε that is not depicted. Moreover, we denote by N^{e}, resp. N^{x}, the set of entry, resp. exit, places of N.

Let N_1 and N_2 be two Petri nets with control flow, we consider four ABCD's control flow operations — that are shown in Figure 1.6 where all the transition guards are True and all the depicted arcs are labelled by $\{\bullet\}$:

- sequential composition $N_1 \fatsemi N_2$ allows to execute N_1 followed by N_2;
- choice $N_1 \Box N_2$ allows to execute either N_1 or N_2;
- iteration $N_1 \circledast N_2$ allows to execute N_1 repeatedly (including zero time), and then N_2 once;
- parallel composition $N_1 \| N_2$ allows to execute both N_1 and N_2 concurrently.

Processes are built on top of atoms comprising either named sub-processes, or (atomic) actions, *i.e.* conditional accesses to typed buffers. Actions may produce to a buffer, consume from a buffer, or test for the presence of a value in a buffer, and are only executed if the given condition is met. The semantics of an action is a transition in a Petri net.

(b) Informal Syntax and semantics of ABCD

ABCD is a compromise between simplicity and expressiveness: the language is useful for many practical situations. In particular, ABCD is well suited to specify modular systems with basic control flow (including concurrency), and possibly complex data handling. This is the case for many examples from the domain of computing; for instance, security protocols will be addressed in Chapter 4. The Formal definition of ABCD is given in [179].

An ABCD specification is an expression composed of the following elements:

1. A possibly empty series of *declarations*, each can be:

   - a function declaration or module import; this corresponds to extensions of the colour domain; the true implementation used the Python language;
   - a buffer declaration; a buffer corresponds to a named place in the semantics, thus buffers are typed, unordered and unbounded;
   - a sub-net declaration; this allows to declare a parametrised sub-system that can be instantiated inside an ABCD term. The language also allows its users to name valid processes into a net declaration and instantiate them repeatedly.

2. A *process term* that plays the role of the "main" process; the semantics of the full specification is that of this term — built in the context of the provided declarations. Process terms are composed of atomic actions or sub-nets instantiations composed with the control flow operators defined above — but replaced with symbols available on a keyboard:

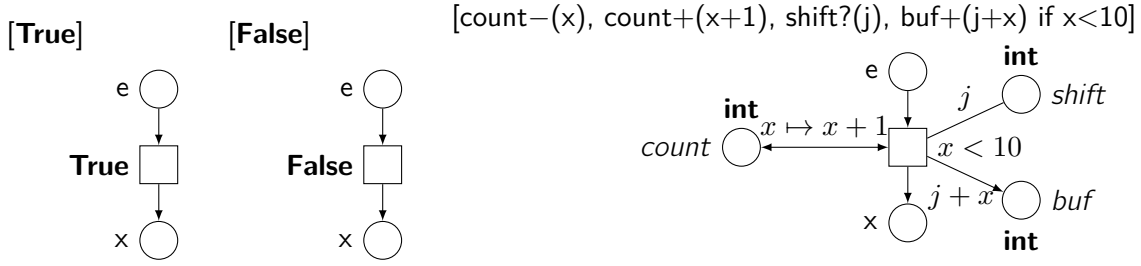


Figure 1.7. The Petri net semantics of various ABCD atomic actions. The undirected arc attached to place *shift* is a read arc that binds j to a token but does not consume it upon firing.

⨟ for sequence, ⊛ for iteration, + for choice and ⊛ |⊛ for parallel. An atomic term is composed of a list of buffer accesses and a guard. For instance:

- [**True**] is the silent action that can always be executed and performs no buffer access;
- [**False**] is the deadlocked action that can never be executed;
- [count−(x), count+(x+1), shift?(j), buf+(j+x) if x<10] is an atomic action that consumes a token from a buffer count binding its value to variable x, produces a token computed as x+1 in the same buffer, binds one token in a buffer shift to variable y without consuming it, and produces the result of j+x in a buffer buf. All this is performed atomically and only if x<10 evaluates to **True**.

The Petri net semantics of these three actions is depicted in Figure 1.7.

A sub-net instantiation is provided as the name of the declared **net** block followed by the list of effective parameters inside parentheses. The semantics of this instantiation is obtained by substituting the parameters inside the definition of the sub-net and building its semantics recursively. Then, the buffers declared locally to the sub-net are made anonymous using the buffer hiding operator.

The semantics of a term is obtained by composing the semantics of its sub-terms using the specified control flow operators. Finally, the semantics of a full specification also includes the initial marking of entry and buffer places.

Like in Python, blocks nesting is defined through source code indentation only, and comments start with character "#" and end with the line. Like in most compiled programming languages (and unlike in Python), ABCD has lexical scoping of the declared elements: an element is visible from the line that follows its declaration until the end of the block in which it is declared. As usual, declaring again an element results in masking the previous declaration.

(c) A simple example

As a very basic example, consider the following producer/consumer example:

```
buffer shared : int = ()
buffer count : int = (1)
[count−(n),count+(n+1),shared+(n)]⊛ [False]
| [shared−(n) if n % 2 == 0]⊛ [False]
```

The [**False**] action is one which can never be executed. The "-" operation on a buffer attempts to consume a value from it and bind it to the given variable, scoped to the current action. The language supplies a read-only version "?", thus count?(n) will read a value from count into variable n without removing it from the buffer. Similarly, the "+" operation attempts to write a value to the buffer, and there are also flush (») and fill («) operations which perform writes into and reads from lists respectively. The first component of the parallel composition above therefore continuously populates the buffer named shared with increasing integers. The second

sub-process just pulls the even ones out of the shared buffer. The Petri net resulting from this ABCD specification is draw in Figure 1.7. Note that for the example shown above, compute the state marking of the generated Petri net with its initial marking would not terminate because the marking graph is infinite. Therefore care must be taken by the ABCD user to ensure that his system has finitely many markings.

As explain before, the declaration of net is modulare. It is thus possible to declare different nets and compose them. A sub-process may be declared as a "net" and reused later in a process expression. That is:

```
net process1():
    buffer state: int = ()
    ...

net process2():
    buffer state: int = ()
    ...
```

then a full system can be specified by running in parallel two instance (in sequence) of the first process and one of the second one:

```
(process1; process1) || process2
```

Typed buffers may also be declared globally to a specification or locally to net processes. For illustring this, we we take another time for example the producer/consumers specification. The producer puts in a buffer "bag" the integers ranging from 0 to 9. To do so, it uses a counter "count" that is repeatedly incremented until it reaches value 10, which allows to exit the loop. The first consumer consumes only odd values from the buffer, the second one consumes only even values. Both never stop looping.

```
def bag : int = () # buffer of integers declared empty

net prod :
    def count : int = 0 # buffer of integers initialised with the single value 0
     [count−(x), count+(x+1), bag+(x) if x < 10] ⊛ [count−(x) if x == 10]

net odd :
  [bag−(x) if (x % 2) == 1] ⊛ [False]

net even :
  [bag−(x) if (x % 2) == 0] ⊛ [False]

prod || odd || even
```

A sub-part of the Petri net resulting from this ABCD specification is draw in Figure. 1.7. It is interesting to note that parts of this ABCD specification are actually Python code and could be arbitrarily complex: initial values of buffers ("()" and "0"); buffer accesses parameters ("x" and "x+1"); actions guards ("x<10", "(x%2)==1", *etc.*).

(d) From ABCD to Coloured Petri nets

To transform ABCD expressions into Coloured Petri nets, the control flow operators are defined by two successive phases given below. Their formal definitions is given in [179].

The first one is a *gluing phase* that combines operand nets and atomic actions; in order to provide a unique definition of this phase for all the operators, we use the *operator nets* depicted in Figure 1.6 to guide the gluing. These operator nets are themselves Petri nets with control

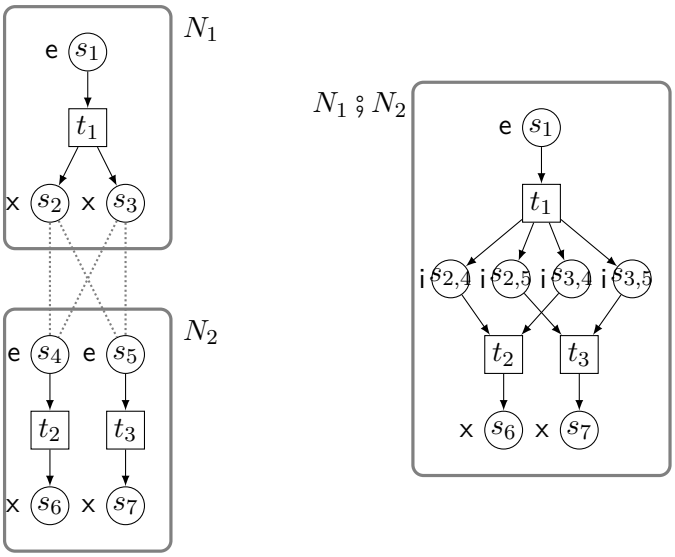


Figure 1.8. On the left: two Petri nets with control flow N_1, N_2. On the right: the result of the gluing phase of the sequential composition $N_1 \fatsemi N_2$. Place names are indicated inside places. Dotted lines indicate how control flow places are paired by the Cartesian product during the gluing phase. Notice that, because no named place is present, the right net is exactly $N_1 \fatsemi N_2$.

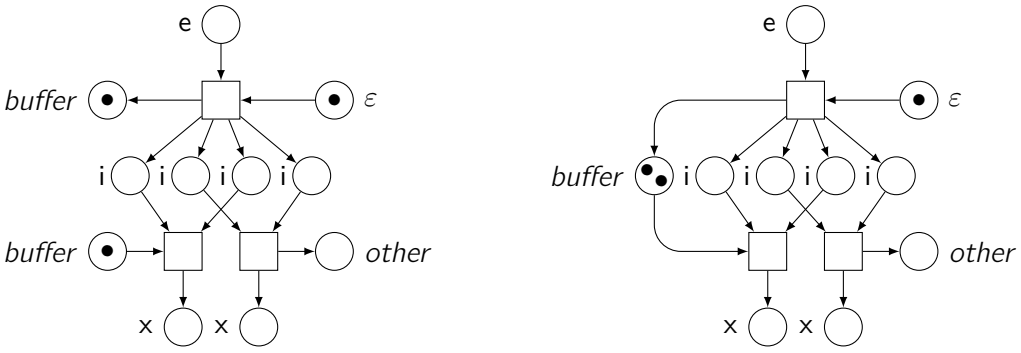


Figure 1.9. On the left, a Petri net with control flow before the merging phase. On the right, named places have been merged.

flow. The second phase is a *named places merging* that fuses the places sharing the same named status.

The intuition behind this glue phase is that each transition of the involved operator net represents one of the operands of the composition. The places in the operator net enforce the correct control flow between the transitions. In order to reproduce this control flow between the composed Petri nets, we have to combine their control flow places in a way that corresponds to what specified in the operator net. An exemple is given in Figure 1.8.

The second phase is a merged name phase. Named places are often used as buffrrs to support asynchronous communication, the sharing of buffers between sub-systems is achieved by the automatic merging of places that share the same name when the sub-systems are composed. In this context, we need a mechanism to specify that a buffer is local to some sub-system. This is provided by the name hiding operation that replaces a buffer name by ε thus forbidding further merges of the corresponding place. Name hiding itself is a special case of a more general status renaming operation. Figure 1.9 shows an examples of this phase.

(e) Structural Information for verification

Providing structural information about the Petri nets to analyse is usually either left to the user of a tool, or obtained by static analysis (*e.g.* place invariants may be computed automatically). However, in our framework, Petri nets are usually constructed by composing smaller parts instead of being provided as a whole. This is the case in particular when the Petri net is obtained as the semantics of a syntax. In such a case, we can derive automatically many structural information about the Petri nets.

For instance, when considering the modelling and verification of security protocols, systems mainly consist of a set of sequential processes composed in parallel and communicating through a shared buffer that models the network. In such a system, we known that, by construction, the set of control flow places of each parallel component forms a 1-invariant, *i.e.* there exist everytime at most one entry place and one exit place. This property comes from the fact that the process is sequential and that the Petri net is control-safe[2] by construction. Moreover, we also know that control flow places are 1-bounded, so we can implement their marking with a Boolean instead of an integer to count the tokens as explained above. It is also possible to analyse buffer accesses at a syntactical level and discover buffers that are actually 1-bounded, for instance if any access is always composed either of a put and a get, or of a test, in the same atomic action.

1.3 Parallelisation

1.3.1 What is parallelism?

A more complete presentation is available at [204].

Many applications require more compute power than provided by sequential computers, like for example, numerical simulations in industry and research, commercial applications such as query processing, data mining and multi-media applications. One option to improve performance is parallel processing.

A parallel computer or multi-processor system is a computer utilizing more than one processor. It's commom to classify parallel computers by distinguishing them by the way how processors can access the system's main memory. Indeed, this influences heavily the usage and programming of the system. Two major classes of distributed memory computers can be distinguished:the distributed memory and the shared memory systems.

(a) Flynn

Flynn defines a classification of computer architectures, based upon the number of concurrent instruction (or control) and data streams available in the architecture [80, 95].

	Single Instruction	Multiple Instructions
Single datum	**SISD**	**MISD**
Multiple data	**SIMD**	**MIMD**

where:

- SISD is "Single Instruction, Single Data stream" that is a sequential machine;
- SIMD is "Single Instruction, Multiple Data streams" that is mostly array processors and GPU;

[2]Let us call control-safe a Petri net with control flow whose control flow places remain marked by at most one token under any evolution. Then, if two Petri nets are control-safe, their composition by any of the control flow operations is also control-safe. This property holds assuming a restriction about how operators may be nested [179].

- MISD is "Multiple Instruction, Single Data stream" that is pipeline of data (pipe skeleton);
- MIMD is "Multiple Instruction, Multiple Data streams" that is clusters of CPUs.

The distributed memory called No Remote Memory Access (NORMA) computers do not have any special hardware support to access another node's local memory directly. The nodes are only connected through a computer network. Processors obtain data from remote memory only by exchanging messages over this network between processes on the requesting and the supplying node. Computers in this class are sometimes also called Network Of Workstations (NOW). And in case of shared memory systems, Remote Memory Access (RMA) computers allow to access remote memory via specialized operations implemented by hardware, however the hardware does not provide a global address space. The major advantage of distributed memory systems is their ability to scale to a very large number of nodes. In contrast, a shared memory architecture provides (in hardware) a global address space, *i.e.* all memory locations can be accessed via usual load and store operations. Thereby, such a system is much easier to program. Also note that the shared memory systems can only be scaled to moderate numbers of processors.

We concentrate on Multiple Instruction, Multiple Data streams (MIMD) model, and especially on one of its subcategory, the so-called Single Program Multiple Data (SPMD) model, wich is the most current for the programmation of parallel computers.

(b) SPMD model

In the SPMD model, the same program runs on each processor but it computes on different parts of the data which were distributed over the processors.

There are two main programming models, message passing and shared memory, offering different features for implementing applications parallelized by domain decomposition. Shared memory allows multiple processes to read and write data from the same location. Message passing is another way for processes to communicate: each process can send messages to other processes

(c) Shared Memory Model

In shared memory model, programs start as a single process known as a master thread that executes on a single processor. The programmer designates parallel regions in the program. When the master thread reaches a parallel region, a fork operation is executed that creates a team of threads, which execute the parallel region on multiple processors. At the end of the parallel region, a join operation terminates the team of threads, leaving only the master thread to continue on a single processor.

In the shared memory model, a first parallel version is relatively easy to implement and can be incrementally tuned. In the message passing model instead, the program can be tested only after finishing the full implementation. Subsequent tuning by adapting the domain decomposition is usually time consuming. We give some well known examples of library for the shared memory programmaing model. OpenMP[1] [24, 47] is a directive-based programming interface for the shared memory programming model. It consists of a set of directives and runtime routines for Fortran, and a corresponding set of pragmas for C and C++ (1998). Directives are special comments that are interpreted by the compiler. Directives have the advantage that the code is still a sequential code that can be executed on sequential machines (by ignoring the directives/pragmas) and therefore there is no need to maintain separate sequential and parallel versions.

Intel Threading Building Blocks (Intel TBB[2]) library [144] which is a library in C++ language that supports scalable parallel programming. The evaluation is done specifically for the pipeline applications that are implemented using filter and pipeline class provided by the library. Various

features of the library which help during pipeline application development are evaluated. Different applications are developed using the library and are evaluated in terms of their usability and expressibility [137]. Recent several years have seen a quick adoption of Graphic Processing Units (GPU) in high performance computing, thanks to their tremendous computing power, favorable cost effectiveness, and energy efficiency. The Compute Unified Device Architecture (CUDA)[3] [164] has enabled graphics processors to be explicitly programmed as general-purpose shared-memory multi-core processors with a high level of parallelism. In recent years, graphics processing units (GPUs) have been progressively and rapidly advancing from being specialized fixed-function to being highly programmable and incredibly parallel computing devices. With the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively programmed using graphics APIs. In CUDA, a GPU can be exposed to the programmer as a set of general-purpose shared-memory Single Instruction Multiple Data (SIMD) multi-core processors. The number of threads that can be executed in parallel on such devices is currently in the order of hundreds and is expected to multiply soon. Many applications that are not yet able to achieve satisfactory performance on CPUs can get benefit from the massive parallelism provided by such devices.

(d) Message Passing Interface (MPI)

The message passing model is based on a set of processes with private data structures. Processes communicate by exchanging messages with special send and receive operations. It is a natural fit for programming distributed memory machines but also can be used on shared memory computers. The most popular message passing technology is the Message Passing Interface (MPI) [189], a message passing library for C and Fortran. MPI is an industry standard and is implemented on a wide range of parallel computers, from multiprocessor to cluster architectures Details of the underlying network protocols and infrastructure are hidden from the programmer. This helps achieve MPI's portability mandate while enabling programmers to focus on writing parallel code rather than networking code. It includes routines for point-to-point communication, collective communication, one-sided communication, parallel IO, and dynamic task creation.

(e) Skeleton paradigm

Anyone can observe that many parallel algorithms can be characterised and classified by their adherence to a small number of generic patterns of computation — farm, pipe, *etc.* Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit with specifications which transcend architectural variations but implementations which recognise them to enhance performance [60]. The core principle of skeletal programming is conceptually straightforward. Its simplicity is its strength.

A well know disadvantage of skeleton languages is that the only admitted parallelism is usually that of skeletons while many parallel applications are not obviously expressible as instances of skeletons. Skeletons languages must be constructed as to allow the integration of skeletal and ad-hoc parallelism in a well defined way [60].

(f) Hybrid architecures and models

Clusters have become the de-facto standard in parallel processing due to their high performance to price ratio. SMP clusters are also gaining on popularity, mainly under the assumption of fast interconnection networks and memory buses. SMP clusters can be thought of as an hierarchical two-level parallel architecture, since they combine features of shared and distributed memory machines. As a consequence, there is an active research interest in hybrid parallel programming models, *e.g.* models that perform communication both through message passing and memory access. Intuitively, a parallel paradigm that uses memory access for intra-node communication and message passing for internode communication seems to exploit better the characteristics of

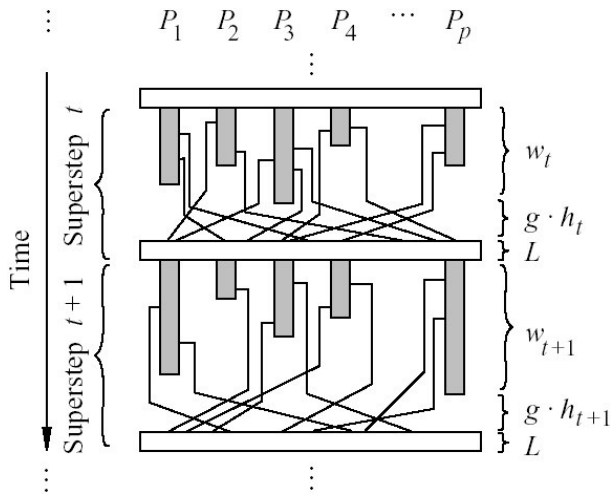


Figure 1.10. The BSP model of execution

an SMP cluster [79] . The hybrid model has already been applied to real scientific applications [123], including probabilistic model checking [120].

1.3.2 Bulk-Synchronous Parallelism

(a) Bulk-Synchronous Parallel Machines

A BSP computer has three components:

- homogeneous set of uniform processor-memory pairs;
- communication network allowing inter processor delivery of messages;
- global synchronization unit which executes collective requests for synchronization barrier.

A wide range of actual architectures can be seen as BSP computers. For example share memory machines could be used in a way such as each processor only accesses a subpart of the shared memory (which is then "private") and communications could be performed using a dedicated part of the shared memory. Moreover the synchronization unit is very rarely a hardware but rather a software — [127] presents global synchronization barrier algorithms. Supercomputers, clusters of PCs, multi-core [110] and GPUs *etc.* can be thus considered as BSP computers.

(b) The BSP's execution model

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed disjoint phases (see Figure 1.10):

1. Each processor only uses its local data to perform sequential computations and to request data transfers to/from other nodes;

2. The network delivers the requested data;

3. A global (collective) synchronisation barrier occurs, making the transferred data available for the next super-step.

(c) BSP's cost model

The performance of the BSP machine is characterised by 4 parameters:

1. the local processing speed **r**;
2. the number of processor **p**;
3. the time **L** required for a barrier;
4. and the time **g** for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word.

The network can deliver an h-relation (every processor receives/sends at most h words) in time $\mathbf{g} \times h$. To accurately estimate the execution time of a BSP program these 4 parameters could be easily benchmarked [29].

The execution time (cost) of a super-step s is the sum of the maximal of the local processing, the data delivery and the global synchronisation times. It is expressed by the following formula:

$$\mathrm{Cost}(s) = \max_{0 \leq i < \mathbf{p}} w_i^s + \max_{0 \leq i < \mathbf{p}} h_i^s \times \mathbf{g} + \mathbf{L}$$

where w_i^s = local processing time on processor i during superstep s and h_i^s is the maximal number of words transmitted or received by processor i during superstep s.

The total cost (execution time) of a BSP program is the sum of its S super-steps costs that is $\sum_s \mathrm{Cost}(s)$. It is, therefore, a sum of 3 terms:

$$W + H \times \mathbf{g} + S \times \mathbf{L} \text{ where } \left\{ \begin{array}{rcl} W & = & \sum_s \max_i w_i^s \\ H & = & \sum_s \max_i h_i^s \end{array} \right.$$

In general, W, H and S are functions of **p** and of the size of data n, or of more complex parameters like data skew. To minimize execution time, the BSP algorithm design must jointly minimize the number S of supersteps, the total volume h and imbalance of communication and the total volume W and imbalance of local computation.

(d) Advantages and inconvienients

As stated in [74]: "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures between the late eighties and the time from the mid-nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain".

This model of parallelism enforces a strict separation of communication and computation: during a super-step, no communication between the processors is allowed, only at the synchronisation barrier they are able to exchange information. This execution policy has two main advantages: first, it removes non-determinism and guarantees the absence of deadlocks; second, it allows for an accurate model of performance prediction based on the throughput and latency of the interconnection network, and on the speed of processors. This performance prediction model can even be used online to dynamically make decisions, for instance choose whether to communicate in order to re-balance data, or to continue an unbalanced computation.

However, on most of cheaper distributed architectures, barriers are often expensive when the number of processors dramatically increases — more than 10 000. But proprietary architectures and future shared memory architecture developments (such as multi-cores and GPUs) make them much faster. Furthermore, barriers have also a number of attractions: it is harder to introduce the possibility of deadlock or livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance [187].

The BSP model considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug since there are many simultaneous communication actions in a parallel program, and their interactions are typically complex. Bulk sending also provides better performances since, from an implementation point of view, grouping communication together in a seperate program phase permits a global optimization of the data exchange by the communications library.

The simplicity and yet efficiency of the BSP model makes it a good framework for teaching (few hours are needed to teach BSP programming and algorithms), low level model for multicores/GPUs system optimisations [110], *etc.*, since it has been conceived has a bridging model for parallel computation.

This is also merely the most visible aspects of a parallel model that shifts the responsibility for timing and synchronization issues from the applications to the communications library[3]. As with other low/high level design decisions, the applications programmer gains simplicity but gives up some flexibility and performance. In fact, the performance issue is not as simple as it seems: while a skilled programmer can in principle always produce more efficient code with a low-level tool (be it message passing or assembly language), it is not at all evident that a real-life program, produced in a finite amount of time, can actually realize that theoretical advantage, especially when the program is to be used on a wide range of machines [114, 147].

Last advantage of BSP is that it greatly facilitates debugging. The computations going on during a superstep are completely independent and can thus be debugged independently. Moreover, it is easy to measure during the execution of a BSP program, time speding to communicate and to synchronise by just adding chronos before and after the primitive of synchronisation. This facility will be used here to compare differents algorithms.

All this capacities are is possibles only because the runtime system knows precisely which computations are independent. In a asynchronous message-passing system as MPI, the independent sections tend to be smaller, and identifying them is much harder. But, using BSP, programers and designer have to keep in mind that some parallelism patterns are not really BSP friendly. For example, BSP does not enjoy in an optimist manner pipeline and master/slave (also knowed as farm of processes) schemes even if it is possible to have not too inefficient BSP programs from these schemes [103]. Thus, some parallel computations and optmisations would never be BSP. This is the drawback of all the restricted models of computations as well.

The BSP model has been used with success in a wide variety of problems such scientific computing [14, 29, 30, 76, 134, 196], parallel data-structure [108, 116], genetic algorithms [39] and genetic programming [78], neural network [182], parallel data-bases [15–17], constraints solver [115], graphs [46, 94, 146, 196], geometry [75], string search [93, 143], implementation of tree skeletons [156], *etc.*

1.3.3 Other models of parallel computation

We survey here, other groups of parallel abstract machines than BSP: the PRAM and derived family, the LogP and extensions of the BSP models.

(a) PRAM

The PRAM (Parallel Random Access Machine) model [97] was introduced as a model for general purpose computations. A PRAM is made by an unbounded number P of processors, each one being a RAM [99] with set of registers rather than a memory. There is only one memory space shared by the processors. In every cycle each processor can perform one of the following actions; read a value from a global memory, write a value from its register to the memory or compute an operation on its local registers. The cost of a random access to the global

[3]BSP libraries are generally implemented using MPI [188] or low level routines of the given specifics architectures

memory is a unit-time independently from the access pattern. The PRAM model can be defined according to the politics to the simultaneously access the same memory localtion, the possible choices are: EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Concurrent Write), CRCW (Concurrent Read Concurrent Write). When a concurrent write is allowed some options are distinguished. The idealization provided by the PRAM completely hides aspects such as synchronization, data locality *etc.* This facilitated the acceptance of the model amongst theorists of algorithms and many parallel algorithms are expressed by using such a abstraction. In practice PRAM has only the P parameter (the number of processors) and the measure of the work performed by an algorithm is simply the time per processor product. Issues such as the communication latency or brandwidth are not considered and this leads to a completely unreliable prediction of execution costs.

The only possibility for hiding (partially) this problem is the exploitation of a certain amount of parallelism to mask the wait for messages. This method is named parallel slackness. Many works has been done to emulate PRAM and, even if, optical technology may turn out a decisive help *etc.*

(b) APRAM

The asynchronous variant of PRAM [111] is intended to be closer to the MIMD architectures. All the models belonging to this family share the needs for an explicit synchronization to ensure that a memory access has been completed. There are two groups of asynchronous PRAM: the phase and the subset. The models of the first group require that all of the processors participate to the synchronization while the models of the second require only a subset. The LPRAM is a variant of the APRAM wich introduces, for the first time the notion of synchronization an latency costs. The cost of synchronizing is considered a nondecreasing function of the processors count: $B(P)$. The latency introduces by the LPRAM is simply a constant d.

(c) HPRAM

The Hierarchical PRAM [125] proposed by Heywood is given by a collection of synchronous PRAM that operate asynchronously from each other. HPRAM can execute a partition instruction to create disjoint subsets of a P processors PRAM. Each subset receives an algorithm to be executed and the partition controls the asynchrony of the model. The model has two variants: (1) Private HPRAM where partition divides memory among processors. In this case, each subset has its own private block of shared memory. Each block is disjoint from the others belonging to the other sub-PRAMs. (2) Shared HPRAM where partition does not divide the shared memory and each sub-PRAM can access the global memory. The parameters of HPRAM are latency $l()$ and the synchronization $s()$. The crucial point is that latency is proportional ot the diameter of the sub-network. In HPRAM there are two different synchronizations: α synchronization wich occurs between processors within a (sub)-PRAM computation and β synchronization wich takes place at the end of a partition. Costs of synchronization are often dominated by communications and computations.

(d) LogP

The LogP [71] model is an asynchronous (or loosely synchronous) framework to describe distributed memory multicomputers which communicate point-to-point. The model provides a vision of the communication costs without taking into account the topology of the interconnection network. The parameters for a LogP machine are: $L($ latency): the upper bound on the latency of the interconnection network, o (overhead): the overhead for transmission or reception (the processor can not overlap this time with other operations), g (gap): the minimum gap between consecutive messages (the reciprocal of g is the bandwidth per processor), P (processors) the number of processors (every local operation is computed in a unit-time named cycle).

(e) CLUMPS

The CLUMPS [41–43] model has been introduced by Campbell as an architectural model (in the McColl [157] classification) wich unifies the characteristic of HPRAM and LogP. The architectural elements used to model a generaliez parallel hardware are: a set of processor-memory pairs, a partionable interconnection network and a flexible control among SIMD and MIMD. Since its partitionable nature, CLUMPS complicates the LogP model by introducing a "regional" rule to compute the values of its main parameters. CLUMPS is the first model which claims to be skeletons-oriented. Unfortunately its cost model fails in providing manageable prediction and the complexity of its performance equations is very high.

Models as HPRAM or CLUMPS can be considered as too "realistic" and this means that they can not be a useful platform for an optimizing tool which aims to be also simple. Moreover many of the current parallel machines do not require such a fine-grain model.

(f) LoPC

The LoPC model [98] has been introduced with the aim of extending the LogP model to account for node-contention. The authors claim that for both fine-grain message passing algorithms and shared memory, the cost due to accesses contention dominates the cost of handlers service time and network latency. The main assumption in the LoPC model are that hardware message buffers at the nodes are infinitely large and that the interconnect is contention free. The model assumes fixed size of the message even if it recalls the LogGP model is made for a possible extension for long messages. The goal of LoPC is to generate a contention efficient runtime scheduling of communications exploiting a limitede set of algorithmic and architectural parameters.

(g) E-BSP

The E-BSP [138] extends the basic BSP model to deal with unbalanced communication patterns *i.e.* patterns in which the amount of data sent or received by each node is different. The cost function supplied by E-BSP is a nonlinear function that strongly depends on the network topology. The model essentially differentiates between communication patterns that are insensitive to the bisection bandwidth and those that are not.

(h) D-BSP

The Decomposable-BSP model [73] extends the BSP model by introducing the possibility of submachine synchronizations. A D-BSP computer is basically a BSP computer where the synchronization device allows subgroup of processors to synchronize independently. The D-BSP remembers the HPRAM and CLUMPS model in which the cost are expressed in terms of BSP supersteps. In this framework network locality can be exploited assuming that submachines parameters are a decreasing functions of the diameter of the subset of processors involved in communication and synchronization.

(i) QSM

Gibbons et *al.* considered the possibility of providing a bridging model based on a shared memory abstraction, in analogy to the message passing based BSP model. The paper introduces the Queuing Shared Model (QSM) [112] wich accounts for bandwidth limitation in the context of a shared memory architecture. The processors execute a sequence of synchronized phases, each consisting of an arbitrary interleaving of the following operations: shared-memory reads, shared-memory writes and local computation.

(j) Multi-BSP

Multi-core architectures, based on many processors and associated local caches or memories, are attractive devices given current technological possibilities, and known physical limitations. Multi-BSP model [200] is a multi-level model that has explicit parameters for processor numbers, memory/cache sizes, communication costs, and syn- chronization costs. The lowest level corresponds to shared memory or the PRAM, acknowledging the relevance of that model for whatever limitations on memory and processor numbers it may be efficacious to emulate it. The Multi-BSP model which extends BSP in two ways. First, it is a hierarchical model, with an arbitrary number of levels. It recognizes the physical realities of multiple memory and cache levels both within single chips as well as in multi-chip architectures. The aim is to model all levels of an architecture together, even possibly for whole datacenters. Second, at each level, Multi-BSP incorporates memory size as a further parameter. After all, it is the physical limitation on the amount of memory that can be accessed in a fixed amount of time from the physical location of a processor that creates the need for multiple levels. An instance of a Multi-BSP is a tree structure of nested components where the lowest level or leaf components are processors and every other level contains some storage capacity. The model does not distinguish memory from cache as such, but does assume certain properties of it.

(k) HiHCoHP

We interest now in a recent model: the Hierarchical Hyper Clusters of Heterogeneous Processors (HiHCoHP) model [44, 45]. It is a successor of the homogeneous LogP model and its long-message extension LogGP. It strives to incorporate enough architectural detail to produce results that are relevant to users of actual (hyper)clusters, while abstracting away enough detail to be algorithmically and mathematically tractable. It intends to be a general-purpose algorithmic model — like logP and logGP.

The HiHCoHP model is rather detailed, exposing architectural features such as the bandwidth and transit costs of both networks and their ports.

Our choice in favor BSP forthe ease of use. Our implementation take into account at this time the architectures NOW in SPMD but scheduled optimizations are easy for hybrid architectures.

1.4 Verifying security protocols

The first class of tools, which focus on verification, typically rely on encoding protocols as Horn clauses and applying resolution-based theorem proving to them (without termination guarantee). Analysis tools of this kind include NRL[4] [158] and ProVerif [31].

We recall that the problem of whether a protocol actually provides the security properties it has been designed for is undecidable [81]. Despite this fact, over the last two decades a wide variety of security protocol analysis tools have been developed that are able to detect attacks on protocols or, in some cases, establish their correctness. We distinguish three classes: tools that attempt verification (proving a protocol correct), those that attempt falsification (finding attacks), and hybrids that attempt to provide both proofs and counterexamples.

1.4.1 Verifying security protocols through theorem proving

One type of mechanized verification process is theorem proving using a higher-order logic theorem prover such as Isabelle/HOL[5] [166, 202] or PVS[6] [169]. Using a theorem prover, one formalizes the system (the agents running the protocol along with the attacker) as a set of possible communication traces. Afterwards, one states and proves theorems expressing that the system in question has certain desirable properties, The proofs are usually carried out under strong restrictions, *e.g.* that all variables are strictly typed and that all keys are atomic.

The main drawback of this approach is that verification is quite time consuming and requires considerable expertise. Moreover, theorem provers provide poor support for error detection when the protocols are flawed.

1.4.2 Verifying security protocols by model checking

The second kind of verification centers around the use of model checkers, which are fully automatic.

In contrast to verification, the second class of tools detects protocol errors (*i.e.* attacks) using model checking [153, 161] or constraint solving [48, 162]. Model checking attempts to find a reachable state where some supposedly secret term is learnt by the intruder, or in which an authentication property fails. Constraint solving uses symbolic representations of classes of such states, using variables that have to satisfy certain constraints. To ensure termination, these tools usually bound the maximum number of runs of the protocol that can be involved in an attack. Therefore, they can only detect attacks that involve no more runs of the protocol than the stated maximum. In the third class, attempts to combine model checking with elements from theorem proving have resulted in backward-search-based model checkers. These use pruning theorems, resulting in hybrid tools that in some cases can establish correctness of a protocol (for an unbounded number of sessions) or yield a counterexample, but for which termination cannot be guaranteed [190].

Model Checking offers a promising approach for automated security analysis of protocols: the intuitive notions are translated into formal specifications, which is essential for a careful design and analysis, and protocol executions can be simulated, making it easier to verify certain security properties. As Model Checking becomes increasingly used in the industry as a part of the design process, there is a constant need for efficient tool support to deal with real-size applications. Model checking [55] is a successful verification method based on reachability analysis (state space exploration) and allows an automatic detection of early design errors in finite-state systems. Model checking works by constructing a model (state space) of the system under design, on which the desired correctness properties are verified.

Model checking is a powerful and automatic technique for verifying finite state concurrent systems. Introduced in early 1980s, it has been applied widely and successfully in practice to verify digital sequential circuit designs and communication protocols. Model checking has been proved to be particularly suited in finding counter-examples, *i.e.* to return paths through the transition system that violate one of the specified system requirements.

At the core of computer security-sensitive applications are security protocols *i.e.* a sequence of message exchanges aiming at distributing data in a cryptographic way to the intended users and providing security guarantees. This leads to the researches in searching for a way to verify whether a system is secure or not. Enumerative model checking is well-adapted to for this kind of asynchronous, non-deterministic systems containing complex data types.

Let us recall that the state space generation problem is the problem of computing the explicit representation of a given model from the implicit one. This space is constructed by exploring all the states (from a function of successor) starting from the initial state. Generally, during this operation, all explored states must be kept in memory in order to avoid multiple exploration of a same state. Once the space is constructed or while an on-the-fly construction, it can be used as input for various verification procedures, such as Linear Temporal Logic (LTL) model-checking. A specialisation of the LTL logic (defined later) to protocols have also be done in [12]. In this document, we will consider the more general problematic of the CTL* logic (defined later) model checking.

State space construction may be very consuming both in terms of memory and execution time: this is the so-called state explosion problem. The generation of large discrete state spaces is so a computationally intensive activity with extreme memory demands, highly irregular behavior, and poor locality of references. This is especially true when complex data-structures are used

in the model as the knowledge of an intruder in security protocols.

During the last decade, different techniques for handling state explosion have been proposed, among which partial orders and symmetries. However these optimizations are not always sufficient. Moreover, most of the currently available verification tools work on sequential machines, which limits the amount of memory and therefore the use of clusters or parallel machines is desirable and is a great challenge of research. As this generation can cause memory thrashing on single or multiple processor systems, it has been lead to consider exploiting the larger memory space available in distributed systems [86,165]. Parallelize the state space construction on several machines is thus done in order to benefit from all the local memories, cpu resources and disks of each machine. This allows to reduce both the amount of memory needed on each machine and the overall execution time.

A distributed memory algorithm with its tool for verification of security protocols is described in [192]. They used buffering principle and also employ a cache of recently sent states in their implementation which task is to decrease the number of sent messages. Unfortunately, the verification of temporal properties is not supported due to the difficulties of combining the parallel checking with the symmetry reduction. We think that extend our algorithm to verify temporal properties would be easy to do.

There are also many "model-checker like" tools dedicated for verifying security protocols as [8, 10, 107]. The most known is certainly the one of [11]. Our approach has the avantage of being general using an algebra of coloured Petri nets and can take into account "protocols with loop" and any data structure using Python.

[101] allows to verify some properties about the protocols for an infinite number of sessions and with some possibility of replay using an algebra of processes. But no logic (LTL or else) can be used here and each time a new property is needed, a new theorem is need to be proved. That can be complicated for the maintenance of the method.

1.4.3 Dedicated tools

A more complete presentation is available at [68].

We firstly recall some earlier approaches relying on general purpose verification tools: Isabelle. Paulson [173] has proposed to state security properties such as secrecy as predicates (formalized in higher-order logic) over execution traces of the protocol, without limitations on the number of agents. These predicates can be verified by induction with automated support provided by the Isabelle proof assistant [166, 202]. Casper[7]/FDR[8]. FDR is a model checker for the CSP process algebra. Roughly speaking, FDR checks whether the behaviors of a CSP process associated with a protocol implementation are included in the behaviors allowed by its specification. FDR is provided with a user-friendly interface for security protocol analysis, Casper [153] that automatically translates protocols in an "Alice & Bob-notation" (with possible annotations) to CSP code. Gavin Lowe has discovered the now well-known attack on the Needham-Schroeder Public-Key Protocol using FDR [152]. Similarly, many protocol-specific case studies have been performed in various general-purpose model checkers. We mention μCRL[9] [34] as used in [35], UPPAAL[10] [25] as used in [63], and SPIN[11] [132] as used in [155].

(a) NRL

In the NRL Protocol Analyzer [158], the protocol steps are represented as conditional rewriting rules. NRL invokes a backward search strategy from some specified insecure state to see if it can be reached from an initial state. It has been used for verification of *e.g.* the Internet Key Exchange protocol [159]. Unfortunately, NRL is not publicly available and it is not clear which condition makes a protocol safe or not.

(b) Athena

The Athena [190] tool is an automatic checking algorithm for security protocols. It is based on the Strand Spaces model [117, 195] and, when terminating, provides either a counterexample if the formula under examination is false, or establishes a proof that the formula is true. Alternatively, Athena can be used with a bound (*e.g.* on the number of runs), in which case termination is guaranteed, but it can guarantee at best that there exist no attacks within the bound. Unfortunately, Athena is also not publicly available.

(c) ProVerif

In ProVerif[12] [31], protocol steps are represented by Horn clauses. The system can handle an unbounded number of sessions of the protocol but performs some approximations — on random numbers. As a consequence, when the system claims that the protocol preserves the secrecy of some value, this is correct; this tool is thus need when no flaw has been found in the protocol (with a bounded number of sessions) and we want to have a test for an unbounded number of sessions. However it can generates false attacks too. Recently an algorithm was developed [1] that attempts to reconstruct attacks, in case the verification procedure fails, adding the possibility of falsification to ProVerif.

(d) LySatool

The LySatool[13] [36] implements security protocol analysis based on a process algebra enhanced with cryptographic constructs. The approach is based on over-approximation techniques and can verify confidentiality and authentication properties.

(e) Constraint solver

Based on [160], in which verification in the Strand Spaces model is translated into a constraint solving problem, an efficient constraint solving method was developed in [64]. The method uses constraint solving, optimized for protocol analysis, and a minimal form of partial order reduction, similar to the one used in [57]. A second version of this tool does not use partial order reduction, enabling it to verify properties of the logic PS-LTL [64].

(f) OFMC

The On-the-Fly Model Checker (OFMC [23]) is part of the AVISPA[14] tool set [11], and is a model checker for security protocols. It combines infinite state forward model checking with the concept of a lazy intruder [21], where terms are generated on-demand during the forward model checking process. A technique called constraint differentiation [162] is employed to avoid exploring similar states in different branches of the search, which is similar to the ideas in [70]. It furthermore supports user-defined algebraic theories [22], allowing for correct modeling of *e.g.* Diffie-Hellman exponentiation.

(g) Scyther

Scyther[15] [68,69] is state-of-the-art in terms of verification speed and provides a number of novel features. (1) It can verify most protocols for an unbounded number of sessions in less than a second. Because no approximation methods are used, all attacks found are actual attacks on the model. (2) In cases where unbounded correctness cannot be determined, the algorithm functions as a classical bounded verification tool, and yields results for a bounded number of sessions. (3) Scyther can give a "complete characterization" of protocol roles, allowing protocol designers to spot unexpected possible behaviours early. (4) Contrary to most other verification tools, the user is not required to provide so-called scenarios for property verification, as all possible protocol

behaviours are explored by default. The Scyther algorithm expands on ideas from the Athena algorithm [191]. A drawback is the impossibility of defining data structures and the limitation of expressivity of the language.

1.5 Model checking

1.5.1 Local (on-the-fly) and global model-checking

In general, one may identify two basic approaches to model-checking. The first one uses a global analysis to determine if a system satisfies a formula; the entire state space of the system is constructed and subjected to analysis. However, these algorithms may be seen to perform unnecessary work: in many cases (especially when a system does not satisfy a specification) only a subset of the system state needs to be analyzed in order to determine whether or not a system satisfies a formula. On the other hand, on-the-fly, or local, approaches to model-checking attempt to take advantage of this observation by constructing the state space in a demand-driven fashion.

For example, the paper [59] presents a local algorithm for model-checking a subpart of the μ-calculus and [201] presents an algorithm for CTL (acronym for Computation Tree Logic) — formally defined latter. [65] gives an algorithm with the same time complexity as the one of [28] for determining when a system satisfies a specification given as a Büchi automaton. In light of the correspondence between such automata and the LTL fragment of CTL* (both formally defined later), it follows that the algorithm from [65] may be used for LTL model-checking also. However, it is not clear how this approach can be extended to handle full CTL* — an exception is the work of [135], apply in [122] on security protocols, where specific game theoric automata are used for verifying on-the-fly CTL* formulas on shared-memory multi-processors but it is also not clear how adapt this method to distributed computations.

Results in an extended version of [26] suggest a model-checking algorithm for full CTL* which allows the on-the-fly construction of the state space of the system. However, this approach requires the *a priori* construction of the states of an amorphous Büchi tree automaton from the formula being checked, and the time complexity is worse than the one of [28].

1.5.2 Temporal logics

Many different modal and temporal logics can serve to express the systems specifications for model checking purposes. A major distinction between temporal logics is whether they see time as linear or branching. This is reflected in the classes of time frames they consider: linear orderings or trees. The formulae of linear time logics are interpreted over linear sequences of actions corresponding to possible runs of the systems. On the other hand, the formulae of branching time logics are interpreted over states (in fact, over computational trees, *i.e.* the structures where the successors of each state are all states reachable from the state in one step). In this thesis we restrict our attention on the linear time logic LTL (Linear Temporal Logic) and the branching time logic CTL* (wich extends the LTL and the Computational Tree Logic CTL) which are both widely used.

1.5.3 Reduction techniques

Contrary to the theorem proving approach, model checking are growing in popularity because it can be partially automated; thus, the verification can be performed within reasonable costs. For this reason many verification tools have have been built. Unfortunately, the model checking have its practical limits that considerably restrict the size of systems that can be verifieed.

This fact is generally referred to as the state space explosion problem. The core of the problem is that the model checking algorithms have to distinguish unexplored states from the explored

ones to prevent their re-exploration. Due to this, they have to maintain a set of already explored states. This set is accessed repeatedly by the algorithms and thus, it has to fit into the main memory of a computer. Otherwise the operating system starts swapping intensively and the computation of a model checking algorithm is practically halted.

Many techniques to fight the limits of enumerative model checkers have been developed to increase model checkers ability. Some techniques are general and some are specific for the given problem. We focus on some important techniques of reduction but first consider the two main approach to build the state space: the explicit and the symbolic ways. The main difference between explicit and symbolic approaches is in the way they maintain and manipulate the set of explored states during the computation.

(a) Explicit Model Checking

The explicit or enumerative model checking algorithms traverse through the state space state by state. Typically, this is ensured by some kind of hashing mechanism. The data structure that implements the set of already visited states has to be optimized for the state presence query and state insertion operations.

(b) Symbolic Model Checking

The symbolic model checking algorithms start either with the set of initial states or with the set of valid states and compute all the successors or predecessors respectively unless the state space is completely generated or an error is found. The standard data structure used to store the states in the symbolic model checking algorithms is the Binary Decision Diagram (BDD) [91, 92]. The BDD structure is capable of storing a large number of the states in a more effcient way than the hash table. However, operations on BDDs are quite complex and dependent on the size of BDD. On the other hand, complexity of BDD operations do not worsen if they are manipulated with sets of states instead of single states. This is why the symbolic model checking algorithms generate the state space not in a state-by-state manner, but in a set-by-set manner. Nevertheless, BDD-based model checking is often still very memory and time consuming. This sometimes circumvents the successful verification of systems. The main reason for the large memory requirements of symbolic model checking is often the huge size of the BDD representing the transition relation. Therefore, some methods have been proposed to diminish this problem [6, 66, 67].

[87–89] present a technic belonging to the family of explicit storage methods wich enables to store the state space in a compact way; this approach being qualified as semi-explicit since all states are not explicitly represented in the state space. Experiments report a memory reduction ratio up to 95% with only a tripling of the computing time in the worst case.

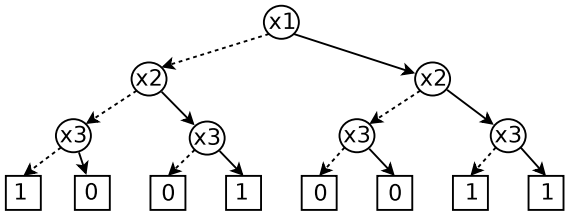


Figure 1.11. a Binary Decision tree.

(c) Abstractions

When the analysis of big models cannot be avoided, it is rarely necessary to consider them in full detail in order to verify or falsify some given property. This idea can be formalized as

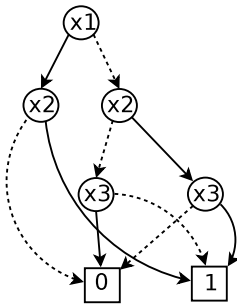


Figure 1.12. A (Reduced) binary decision diagram for the binary decision tree of Figure 1.11.

an abstraction function (or relation) that induces some abstract system model such that the property holds of the original, "concrete" model if it can be proven for the abstract model. Abstraction [54] is used to hide details in the model that are irrelevant to the satisfaction of the verified property, hence reducing the total number of reachable states in the state space. In general, the appropriate abstraction relation depends on the application and has to be defined by the user. Abstraction-based approaches are therefore not entirely automatic "push-button" methods in the same way that standard model checking is.

(d) Partial Order Reduction (POR)

Another popular technique to fight the state explosion problem is partial order reduction (POR) — many POR reduction techniques are described and referenced in [55]. The technique partitions the state space into equivalence classes (using an equivalence relation) and then verify only representative executions for each equivalence class. The method exploits the commutativity of the interleavings of asynchronous processes, because not all the possible interleavings of several asynchronous processes are necessarily needed to establish the correctness of a given property. This technique works well mainly for systems that are made of asynchronous and interleaving components in which case the stuttering equivalence is used to reduce the state space size significantly. There is always a tradeoff between the potential effectiveness of a reduction method and the overhead involved in computing a sufficient set of actions that must be explored at a given state. Moreover, the effectiveness of partial-order reductions in general depends on the structure of the system: while they are useless for tightly synchronized systems, they may dramatically reduce the numbers of states and transitions explored during model checking for loosely coupled, asynchronous systems.

[57, 58] present a POR algorithm for security protocols and determine the class of modal properties that are preserved by it. They observe that the knowledge of the Dolev-Yao attacker model in the course of each protocol run is non-decreasing, and, intuitively, with more knowledge the attacker can do more (harm). Therefore, when verifying security protocols which yield finite-depth executions, in the presence of the Dolev-Yao attacker, it is safe to prioritize actions that increase the attacker's knowledge over other actions.

[96] report on extensions of the POR algorithm of [57, 58] to handle security protocols in which participants may have choice points.

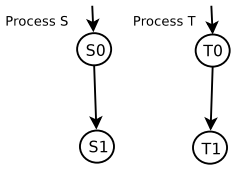


Figure 1.13. Two independent concurrent processes.

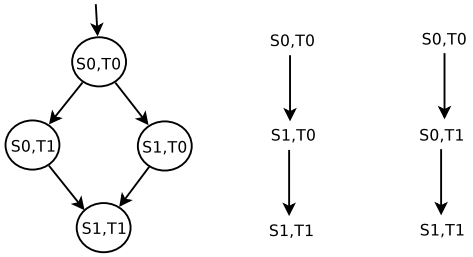


Figure 1.14. Interleavings and execution sequences of the two processes in 1.13 — see [135].

(e) State Caching

In explicit-state model checking the full state space is often stored in a hash table; states are hashed to a specific position in the table and stored there. A classic approach of dealing with collisions in a hash table is to store the multiple states, hashed to the same position, in a linked list.

State caching is a method that uses hash tables, but deals differently with collisions in an attempt to make explicit model checking feasible for a state space that is too large to fit within the available memory. The method, as described in [130], restricts the storage space to a single array of states, *i.e.* no extra linked lists are used. Initially all states are stored in the hash table, but when the table fills up, new states can overwrite old states. This does not effect the correctness of the model checker, but can result in duplicate work if a state, that has been overwritten, is reached again [113]. Runtime may thus increase significantly where the hash table is much smaller than the full state space.

(f) Bitstate Hashing

This technique was also introduced as an alternative hashing technique [131]; it uses a normal hash table, but without collision detection. As described in the previous section, states that are hashed to the same position are typically stored in a linked list. If this technique is used and a state is hashed to a nonempty slot, it is compared with all the states in the linked list to determine whether the hashed state has been visited before. If no collision detection is used and a state is hashed to a nonempty slot it is assumed that the state has been visited; thus only 1 bit is needed per slot to indicate whether a state has been hashed to it or not. The side effect of using only 1 bit is that part of the state space may be ignored, because when more than one state is hashed to the same slot only the first hashed state will be explored.

(g) Probabilistic Techniques

Also probabilistic techniques found their applications in model checking. The random walk method employs probabilistic choice to build random system executions that are examined for presence of an error. If an error is found, the trace of random walk provides the needed counterexample, if not, either more random walks can be executed or the model is declared correct. Due to this the correctness of the system is ensured only with a certain probability. This method is a typical example of an error discovery method and it may be considered much closer to testing than to verification methods. However, there are other probabilistic methods that support model checking algorithms and have nothing common with testing. A good example is a technique that employs probabilistic choices to make the decision of whether to save a state in the set of visited states; thus, trading time for space.

(h) Symmetry

One try to exploit symmetries, which often exist in concurrent systems. It has been shown that in model checking of concurrent systems which exhibit a lot of symmetries often significant memory savings can be achieved (see *e.g.* [136]). Symmetry reduction techniques [5, 53, 84] in verifiation of concurrent systems generally exploit symmetries by restricting statespace search to representatives of equivalence classes. The calculation of the equivalence class representatives is central to all model checking methods which use symmetry reduction. Symmetry reduction can be contrasted with partial order reduction as follows. Partial order reduction considers sets of paths; a set of independent paths from one state to another state is replaced by a single representative path. Symmetry reduction, on the other hand, considers sets of states; a group of equivalent states is replaced by a single representative state. It is known that for arbitrary symmetries their computation is a hard and time-consuming problem. But it has been shown that for certain classes of symmetries this problem can be solved efficiently. Although the state-space can be reduced considerably by using symmetry reduction, their usage can lead to a signficant increase in runtime.

Intuitively, the correctness of the protocol should not depend on the specific assignment of principal names. In other words, by permuting the names of the principals the correctness of the protocol should be preserved.

[53, 56] have also developed the theory of symmetry reductions in the context of verifying security protocols. Intuitively the state space is partitioned into various equivalence classes because of the inherent symmetry present in the system. During the verification process the algorithm only considers one state from each partition.

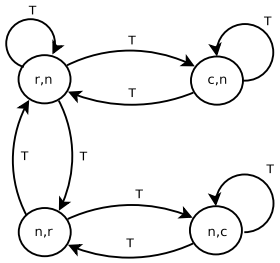


Figure 1.15. A Kripke structure.

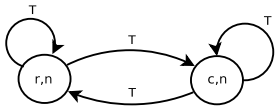


Figure 1.16. The quotient structure for 1.15 — see [53].

(i) Petri Nets Unfoldings

Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. Petri net unfoldings [85, 140] relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic net. Unfoldings provide one way to exploit this observation. An unfolding is a mathematical structure that explicitly represents concurrency and causal dependence between events, and also the points where a choice occurs between qualitatively different behaviors. Like a computation tree, it captures at once all possible behaviors of a system, and we need only examine a finite part of it to answer certain questions about the system. However, unlike a computation tree, it does not make interleavings explicit, and so it can be exponentially more concise.

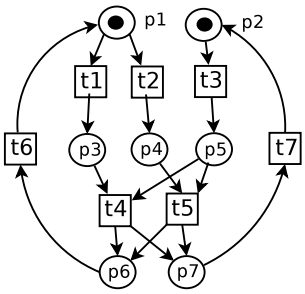


Figure 1.17. A Petri net system.

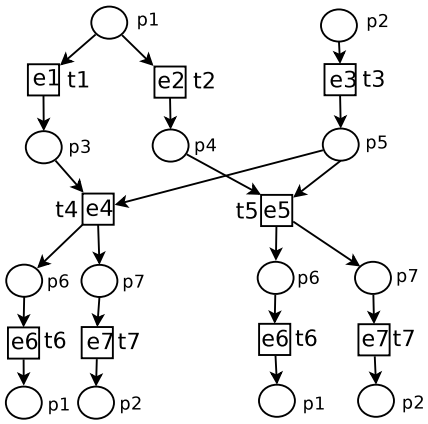


Figure 1.18. An unfolding of the Petri net of the Figure 1.17.

1.5.4 Distributed state space generation

All above mentioned techniques have one common attribute: they try to reduce the state space. Some do that by reducing the number of states in the state space and others by improving data structures used to save the set of visited states. This thesis focuses on a technique that does not reduce the state space, but, contrary to all previously mentioned approaches, increases the available computational and storage power. This technique builds on the idea of storing the state space in a distributed memory environment.

One of the main technical issues in the distributed memory state space generation is a way how to partition the state space among participating workstations. Most of approaches to the distributed memory state space generation use a partitioning mechanism that works at level of states which means that each single state is assigned to a machine and it belongs to. This assignment is done by a partition function that partitions the state space into subsets of states. Each such a subset is then owned by a single workstation.

The main idea of most known approaches to the distributed memory state space generation is similar. The state space generation is started from an initial state by the workstation that owns it (with respect to the partition function). Successors of the initial state are gradually generated. When a successor of a state is generated that belongs to a different workstation, it is wrapped into a message and sent over the network to the owning workstation. As soon as the workstation receives a message containing a state, it starts generating its successors. Those newly generated successors that do not remain local are sent over network and processed by the target workstation in the same manner. This procedure continues until the entire state space is generated and so no messages are sent anymore [102]. To detect this situation a termination detection procedure is usually employed. Furthermore, if a complete static load balancing scheme is considered, the function of partition is typically fixed at the compile-time of the algorithm. Unfortunately, in such a case that ensures well balanced computation but increases excessively

Figure 1.19. Model without parallelism

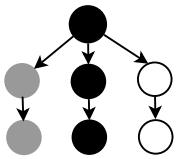


Figure 1.20. Model with limited parallelism

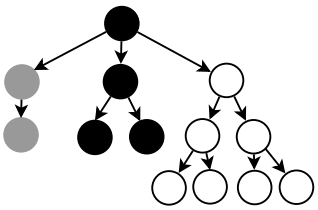


Figure 1.21. Problem of balance

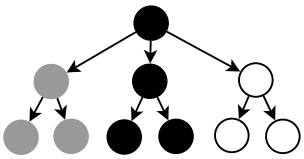


Figure 1.22. Ideal memory distribution

the communication complexity. A slightly different situation is when the partition function is not known at the compile-time, but it is computed once after the initialization. Such a partition function is certainly static as well — *e.g.* [19].

Finite state space construction can be classified as an irregular problem in the parallel algorithms community because of the irregularity of its structure, in other words, the cost to operate this kind of structure is not exactly know or is unknown by advance. As a consequence, the parallel execution of such problems may result in a bad load balance among the processors [91].

In [199], the authors explain that the characteristics of the model under consideration has a key influence on the performance of a parallel algorithm because it may result in extra overhead during the exploration task. Figure 1.19 shows a model where the parallel exploration will perform like a sequential one, incapable of speedups. Figure 1.20 illustrates a model that imposes high scheduling overheads, due to the small size of the work units. The ideal model being where (almost) every node has more than one successor, minimizing the scheduling overhead.

But the main problem is the load balance between the different processors involved in the model-checking procedure. Figure 1.21 shows a high imbalance in the distribution of states across the processors — represented in differents colors. Figure 1.22 is an ideal memory distribution accross the machines and during the generation — for three processors.

Distributed state space construction has been studied in various contexts. All these approaches share a common idea: each machine in the network explores a subset of the state space. This procedure continues until the entire state space is generated and so no messages are sent anymore [102]. To detect this situation a termination detection procedure is usually employed. However, they differ on a number of design principles and implementation choices such as: the way

of partitioning the state space using either static hash functions or dynamic ones that allow dynamic load balancing, *etc.* In this section, we focuss on some of these technics and discuss their problems and advantages. More references can be found in [18].

To have efficient parallel algorithms for state space generation, we see two requirements. First, the partition function must be computed quickly and so that a child state (from the sucessor function) is likely to be mapped to the same processor as its parent otherwise we will be overwhelmed by inter-processor communications (the so called cross transitions) which obviously implies a drop of the locallity of the computations and thus of the performances. Second, balancing of the workload is obviously needed [145]: the problem of well balanced computation is an inseparable part of the distributed memory computing problem because its help to fully profit from available computational power allowing them to achieve expected speedup. The problem is hampered by the fact that future execution requirements are unknown, and unknowable, because the structure of the undiscovered portion of the state space is not known.

Note that employing dynamic load balancing scheme can be problematic in some cases as it can be difficult to appropriately modify the algorithm that is intended to be used under a dynamic load balancing scheme. While it has been showed that a pure static hash-function for the partition function can effectively balance the workload and achieve reasonable execution time efficiencies as well [102], the method suffers from some obvious drawbacks [18, 170]. First there can be too much number of cross transitions. Second, if ever in the course of the generation, just one processor is so burdened with states that it exhausts its available memory, the whole computation fails or slowing too much due to the virtual memory management of the OS. And it seems impossible for this kind of partition function to find without complex heuristics when states can be save into disks to relax the main memory.

As regards high-level languages for asynchronous concurrency, a distributed state space exploration algorithm [148] derived from the Spin model-checker has been implemented using a work/slave model of computation. Several Spin-specific partition functions are experimented, the most advantageous one being a function that takes into account only a fraction of the state vector. The algorithm performs well on homogeneous networks of machines, but it does not outperform the standard except for problems that do not fit into the main memory of a single machine. In the same manner, in [165] the authors exploit certain characteristics of the system to optimise the generation using first a random walk on the beginning of the space graph. This work has been extend in [186] but it is not clear which models fits well to their heuristics and how apply this to protocols.

Another distributed state enumeration algorithm has been implemented in the Murφ verifier [193]. The speedups obtained are close to linear and the hash function used for state space partition provides a good load balancing. However, experimental data reported concerns relatively small state spaces (approximatively 1.5 M states) on a 32-node UltraSparc Myrinet network of workstations.

There also exist approaches, such as [141], in which parallelization is applied to "partial verification", *i.e.* state enumeration in which some states can be omitted with a low probability. In our project, we only address exact, exhaustive verification issues. For completeness, we can also mention an alternative approach [124] in which symbolic reachability analysis is distributed over a network of workstations: this approach does not handle states individually, but sets of states encoded using BDDs.

For the partition function, different technics has been used. In [102] authors used of a primer number of virtual processors and mapping them to real processor. That improves load balancing but not the problematic of cross transitions. In [174], the partition function is computed by a round-robin of the childs. That improves locallity of the computations but can duplicates states and its works well only when network are slower enought that compute states is much faster than sending them which is not the case on modern architectures. In [167], an user's defined abstract interpretation is used to reduce the size of the state space and then it allows to distribute the

abstract graph following by computing real states fully in parallel. A tool is given for helping the users to find this abstraction. We have not find how this technic can be apply to security protocols.

In [32,33,133] authors used complex distributed file system or shared database to optimise the sending of the states especially when complex data-structure are used internally in the states — as ours. That can improve the implementation but not the idea of our algorithms. The use of saturation for parallel generation is defined in [50] but improve only memory use and does not achieve a clear speedup with respect to a sequential implementation.

For load balacing technics we can cite [2] when remaping is initiated by the master node when the memory utilization of one node differs more than a given percentage from the average utilization of all the others. In the same way, [150] presented a new dynamic partition function scheme that builds a dynamic remapping, based on the fact that the state space is partitioned into more pieces than the number of involved machines. When load on a machine is too high, the machine releases one of the partitions it is assigned and if it is the last machine owning the partition it sends the partition to a less loaded machine. This mechanism reduces the number of messages sent which is done to the detriment of redundant works if a partition is owned by several machines and a partial inconsistence may occur when a partition is moved unless all the other machines are informed about its movement.

In [170, 171] extended differents technics of the literature that tries avoid sending a state away from the current network node if its 2nd-generation successors are local and a mechanism that prevents re-sending already sent states. The idea is to compute latter the state for model-cheking which can be faster than sending it. That clearly improves communications but our technic performs the same job without ignoring any of the states.

In [118, 119] present a generic multithreaded and distributed infrastructure library designed to allow distribution of the model checking procedure over a cluster of machines. This library is generic, and is designed to allow encapsulation of any model checker in order to make it distributed.

1.6 Outline

Since these protocols are at the core of security-sensitive applications in a variety of domains, their proper functioning is crucial as a failure may undermine the customer and, more generally, the public trust in these applications. Designing secure protocols is a challenging problem [20,61]. In spite of their apparent simplicity, they are notoriously error-prone. Surprisingly, severe attacks can be conducted even without breaking cryptography, but by exploiting weaknesses in the protocols themselves, for instance by carrying out man-in-the-middle attacks, where an attacker plays off one protocol participant against another, or replay attacks, where messages from one session (*i.e.* execution of an instance of the protocol) are used in another session. It is thus of utmost importance to have tools and specification languages that support the activity of finding flaws in protocols.

(a) State Space

In this Chapter 2, we exploit the well-structured nature of security protocols and match it to the BSP [29, 187] model of parallel computation. This allows us to simplify the writing of an efficient algorithm for computing the state space of finite protocol sessions. The structure of the protocols is exploited to partition the state space and reduce cross transitions while increasing computation locality. At the same time, the BSP model allows to simplify the detection of the algorithm termination and to load balance the computations.

First, we briefly review in Section 2.1 the context of our work that is models of security protocols and their state space representation as LTS. Section 2.2 describes first attempt of parallelisation that is a naive parallel algorithm for the state space construction. Then, Section 2.3

is dedicated to, in a first time, the hypothesis concerning our protocols model, then in more subtle algorithms increasing local computation time, decreasing local storage by a sweep-line reduction and balancing the computations. Finally, explanations on the appropriateness of our approach are discussed in Section 2.4.

(b) Model Checking

Checking if a cryptographic protocol is secure or not is an undecidable problem [81] and even a NP problem restricted to a bounded number of agents and sessions [183]. However, enumerative model-checking is well-adapted for finding flaws [11] and some results exist by extending bound to unbound number of sessions [7]. In the following, we consider the problem of checking a LTL and CTL* formulas over labelled transition systems (LTS) that model security protocols. Checking a LTL or CTL* formula over a protocol is not new [9, 23, 100, 122, 122] and have the advantage over dedicated tools for protocols to be easily extensible to non standard behaviour of honest principals (*e.g.* contract-signing protocols: participants required to make progress) or to check some security goals that cannot be expressed as reachability properties, *e.g.* fair exchange. A specialisation of LTL to protocols have also be done in [62]. We consider also the more general problematic of CTL* model checking.

The peculiarity of our work concerns the parallel computation. In this Chapter 3, we recall the well known Tarjan algorithm which is the underlying structure of the work on a local approach used by [28] for CTL* model checking. [28] is our working basis and our main contributions in the following sections are essentially the adaptation of the algorithms found in [28] for the parallel case of security protocols.

(c) Case Study

This Chapter 4 concerns the practical part of our work. In a first time, we present the specification of security Protocols by the langage ABCD and we give several examples of protocols with their modelisation in this langage. Then, we describe the important technologies we use to implement our algorithms: the BSP Python Programming library and the SNAKES toolkit and syntactic layers wich is a Python library to define, manipulate and execute coloured Petri nets [178]. Then we give the features of the implementation of our parallel algorithms and at last the benchmarks on our differents algoritms.

2 Stace space

This chapter extends the work of [104].

Contents

2.1 Security protocols as Label Transition System 42

2.1.1 Label Transition System and the marking (state) graph 42

2.1.2 LTS representation of security protocols 42

2.1.3 From LTS to high-level Petri nets 42

2.1.4 Sequential state space algorithm 44

2.2 A naive parallel algorithm 44

2.3 Dedicated parallel algorithms 46

2.3.1 Our generic protocols model 46

2.3.2 Having those structural informations from ABCD models 47

2.3.3 Increasing local computation time 47

2.3.4 Decreasing local storage: sweep-line reduction 49

2.3.5 Balancing the computations 49

2.4 Formal explanations of the LTS hypothesis 51

2.4.1 General assumptions 51

2.4.2 Slices 53

2.4.3 Receptions and classes 54

2.4.4 Termination of the algorithms 55

2.4.5 Balance considerations 55

In this Chapter 2, we exploit the well-structured nature of security protocols and match it to the BSP [29, 187] model of parallel computation. This allows us to simplify the writing of an efficient algorithm for computing the state space of finite protocol sessions. The structure of the protocols is exploited to partition the state space and reduce cross transitions while increasing computation locality. At the same time, the BSP model allows to simplify the detection of the algorithm termination and to load balance the computations.

First, we briefly review in Section 2.1 the context of our work that is models of security protocols and their state space representation as LTS. Section 2.2 describes first attempt of parallelisation that is a naive parallel algorithm for the state space construction. Then, Section 2.3 is dedicated to, in a first time, the hypothesis concerning our protocols model, then in more subtle algorithms increasing local computation time, decreasing local storage by a sweep-line reduction and balancing the computations. Finally, explanations on the appropriateness of our approach are discussed in Section 2.4.

2.1 Security protocols as Label Transition System

2.1.1 Label Transition System and the marking (state) graph

A *labelled transition system* (LTS) is an implicit representation of the state space of a modelled system. It is defined as a tuple (S, T, ℓ) where S is the set of states, $T \subseteq S^2$ is the set of transitions, and ℓ is an arbitrary labelling on $S \cup T$. Given a model defined by its initial state s_0 and its successor function succ, the corresponding explicit LTS is $\mathsf{LTS}(s_0, \mathsf{succ})$, defined as the smallest LTS (S, T, ℓ) such that s_0 in S, and if $s \in S$ then for all $s' \in \mathsf{succ}(s)$ we also have $s' \in S$ and $(s, s') \in T$. The labelling may be arbitrarily chosen, for instance to define properties on states and transitions with respect to which model checking is performed.

In the following, the presented algorithms compute only S. This is made without loss of generality and it is a trivial extension to compute also T and ℓ, assuming for this purpose that $\mathsf{succ}(s)$ returns tuples $(\ell(s, s'), s', \ell(s'))$. This is usually preferred in order to be able to perform model-checking of temporal logic properties.

2.1.2 LTS representation of security protocols

In this thesis, we consider models of security protocols, involving a set of *agents*, given as a labelled transition systems (LTS). We also consider a Dolev-Yao attacker that resides on the network [77]. An execution of such a model is thus a series of message exchanges as follows.

1. An agent sends a message on the network.

2. This message is captured by the attacker that tries to learn from it by recursively decomposing the message or decrypting it when the key to do so is known. Then, the attacker forges all possible messages from newly as well as previously learnt informations (*i.e.* attacker's knowledge). Finally, these messages (including the original one) are made available on the network.

3. The agents waiting for a message reception accept some of the messages forged by the attacker, according to the protocol rules.

Because of undecidability or efficiency concerns, one apply some restrictions on the considering model given as input by limiting the number of agents put into play in the model, this being defined by a Scenario. This restriction induces a finite state space making possible its pratical construction.

2.1.3 From LTS to high-level Petri nets

In this thesis, we consider models of security protocols, involving a set of *agents* where a Dolev-Yao attacker resides on the network. As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* called ABCD [181] allowing for easy and structured modelling. However, our approach is largely independent of the chosen formalism and it is enough to assume that some properties define in [104] hold.

ABCD (Asynchronous Box Calculus with Data [180]) is a specification language that allows its users to express the behavior concurrent systems at a high level. A specification is translated into colored Petri nets. In particular, the ABCD meta syntax allows its users to define a complex processes in an algebra that allows: sequential composition (P;Q); non-deterministic choice (P+Q); iteration (P*Q=Q+(P;Q)+(P;P;Q)+$\cdots$); parallel composition (P$\|$Q). Processes are built on top of atoms comprising either named sub-processes, or (atomic) actions, *i.e.* conditional accesses to typed buffers. Actions may produce to a buffer, consume from a buffer, or test for the presence of a value in a buffer, and are only executed if the given condition is met. The semantics of an action is a transition in a Petri net.

For a description of the syntax and semantics of ABCD, as well as an illustrative example, please consult [180]. As a very basic example, consider the *Woo and Lam* taken from *SPORE* [154]. This protocol ensures one-way authentication of the initiator A of the protocol to a responder B using symmetric-key cryptography and a trusted third-party server S with share long-term symmetric keys and a fresh and unpredictable nonce produced by B:

```
A, B, S  :    principal
Nb       :    nonce
Kas, Kbs :    skey

1.    A  ->  B  :    A
2.    B  ->  A  :    Nb
3.    A  ->  B  :    {Nb}Kas
4.    B  ->  S  :    {A, {Nb}Kas}Kbs
5.    S  ->  B  :    {Nb}Kbs
```

which could be model using ABCD as:

```
net Alice (A, agents, S) :
    buffer B_ : int = ()
    buffer Nb_ : Nonce = ()
    [agents?(B), B_+(B), snd+(A)]                    # 1. ->
    ; [rcv?(Nb), Nb_+(Nb)]                           # 2. <-
    ; [Nb_?(Nb), snd+(("crypt", ("secret", A, S), Nb))] # 3. ->

net Bob (B, S) :
    buffer A_ : int = ()
    buffer myster_ : object = ()
    [rcv?(A), A_+(A)]                  # 1. <-
    ; [snd+(Nonce(B))]                 # 2. ->
    ; [rcv?(myster), myster_+(myster)] # 3. <-
    ; [A_?(A), myster_?(myster),
       snd+(("crypt", ("secret", B, S), A, myster))] # 4. ->
    ; [rcv?(("crypt", ("secret", S, B), Nb))
        if Nb == Nonce(B)]             # 5. <-

net Server (S) :
    buffer B_ : int = ()
    buffer Nb_ : Nonce = ()
    [rcv?(("crypt", ("secret", B, S), A,
      ("crypt", ("secret", A, S), Nb))), B_+(B), Nb_+(Nb)] # 4. <-
    ; [B_?(B), Nb_?(Nb), snd+(("crypt", ("secret", S, B), Nb))] # 5. ->
```

The '-' operation on a buffer attempts to consume a value from it and bind it to the given variable, scoped to the current action. The language also supplies a read-only version '?', thus rcv?(Nb) will read a value from rcv into variable Nb without removing it from the buffer. Similarly, the '+' operation attempts to write a value to the buffer, and there are also flush (») and fill («) operations which perform writes into and reads from lists respectively. Note that we used two buffer called rcv and snd which model the sending and receip in a network. Encoded message are tuple with special values as crypt and secret that attacker agent could not read if he have the keys.

The attacker has three components: a buffer named knowledge which is essentially a list of the information that the attacker currently knows, a list of initial knowledge, and a learning engine with which it uses to glean new knowledge from what it observes on the network. Intuitively, the attacker performs the following operations:

1. it intercepts each me ssage that appears on buffer nw which represent the network and adds it to its knowledge;

2. it passes each message along with its current knowledge to the learning engine and adds any new knowledge learned to it's current knowledge;

3. it then may either do nothing, or take any message that is a valid message in the protocol that is contained in its knowledge and put it back on buffer nw.

```
1 def sequential_construction(s_0) is
2   todo ←{s_0}
3   known ←∅
4   while todo ≠ ∅
5     pick s from todo
6     known ←known ∪ {s}
7     for s' ∈ succ(s) \ known
8       todo ←todo ∪ {s'}
```

Figure 2.1. Sequential construction.

In ABCD, these actions are expressed by the following term:

```
[nw−(m), knowledge>>(k), knowledge<<(learn(m,k))];
[True] + [knowledge?(x), nw+(x) if message(x)]
```

The first line implement steps 1 and 2: a message m is removed from the network, and this message is passed to a method learn() along with the contents of the current knowledge. The return value of this method is filled back into the knowledge buffer. The next line implements step 3 : the process can either choose to do the empty action [True], or to replay any element of its knowledge that satisfies the message() predicate back - which checks if x is a valid protocol message - on the network. Note that a branch is created in the state space for each message that can be intercepted in the first line, another for the choice in the second line, and another for each valid message in the knowledge. This is why the attacker is the most computationally intensive component of our modelling.

As Python's expressions are used in this algebra, the learning engine (the Dolev-Yao inductive rules) is a Python function and could thus be easaly extended for taking account specific properties of hashing or of crypto primitives.

2.1.4 Sequential state space algorithm

In order to explain our parallel algorithm, we start with Algorithm 2.1 that corresponds to the usual sequential construction of a state space. The sequential algorithm involves a set todo of states that is used to hold all the states whose successors have not been constructed yet; initially, it contains only the initial state s_0. Then, each state s from todo is processed in turn and added to a set known while its successors are added to todo unless they are known already. At the end of the computation, known holds all the states reachable from s_0, that is, the state space S.

2.2 A naive parallel algorithm

We now show how the sequential algorithm can be parallelised in BSP and how several successive improvements can be introduced. This results in an algorithm that remains quite simple in its expression but that actually relies on a precise use of a consistent set of observations and algorithmic modifications. We will show in the next section that this algorithm is efficient despite its simplicity.

Algorithm 2.1 can be naively parallelised by using a partition function cpu that returns for each state a processor identifier, *i.e.* the processor numbered cpu(s) is the owner of s. Usually, this function is simply a hash of the considered state modulo the number of processors in the parallel computer. The idea is that each process computes the successors for only the states it owns. This is rendered as Algorithm 2.2; notice that we assume that arguments are passed by references so that they may be modified by sub-programs.

This is a SPMD (Single Program, Multiple Data) algorithm so that each processor executes it. Sets known and todo are still used but become local to each processor and thus provide only

```
1 def main(s_0) is
2   todo ←∅
3   total ←1
4   known ←∅
5   if cpu(s_0) = mypid
6     todo ←todo ∪ {s_0}
7   while total > 0
8     tosend ←successor(known,todo)
9     todo, total ←exchange(known,tosend)
```

```
1 def successor(known, todo) is
2   tosend ←∅
3   while todo ≠ ∅
4     pick s from todo
5     known ←known ∪ {s}
6     for s' ∈ succ(s) \ known
7       if cpu(s') = my_pid
8         todo ←todo ∪ {s'}
9       else
10        tosend ←tosend ∪ {(cpu(s',s')}
11  return tosend
```

```
1 exchange(known, tosend) is
2   received, total ←BSP_EXCHANGE(tosend)
3   return (received \ known),total
```

Figure 2.2. Naive BSP construction.

a partial view on the ongoing computation. So, in order to terminate the algorithm, we use an additional variable total in which we count the total number of states waiting to be proceeded throughout all the processors, *i.e.* total is the sum of the sizes of all the sets todo. Initially, only state s_0 is known and only its owner puts it in its todo set. This is performed in lines 4–6, where my_pid is evaluated locally to each processor to its own identifier.

Function successor is then called to compute the successors of the states in todo. It is essentially the same as the sequential exploration, except that each processor computes only the successors for the states it actually owns. Each computed state that is not owned by the local processor is recorded in a set tosend together with its owner number. This partitioning of states is performed in lines 7–11.

Then, function exchange is responsible for performing the actual communication between processors. The primitive BSP_EXCHANGE sends each state s from a pair (i, s) in tosend to the processor i and returns the set of states received from the other processors, together with the total number of exchanged states. The routine BSP_EXCHANGE performs a global (collective) synchronisation barrier which makes data available for the next super-step so that all the processors are now synchronised. Then, function exchange returns the set of received states that are not yet known locally together with the new value of total. Notice that, by postponing communication, this algorithm allows buffered sending and forbids sending several times the same state.

It can be noted that the value of total may be greater than the intended count of states in todo sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception. Moreover, if this state has been also computed by its owner, it will be ignored. This not a problem in practise because in the next super-step, this duplicated count will disappear. In the worst case, the termination requires one more super-step during which all the processors will process an empty todo, resulting in an empty exchange and thus total=0 on every processor, yielding the termination.

Furthermore, this algorithm allows buffering sending states and forbids sending several time the same state in the same super-step.

2.3 Dedicated parallel algorithms

2.3.1 Our generic protocols model

In this thesis, we consider models of security protocols involving a set of *agents* and we assume that any state can be represented by a function from a set $\mathcal{L}$ of *locations* to an arbitrary data domain $\mathcal{D}$. For instance, locations may correspond to local variables of agents, shared communication buffers, *etc.*

As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* [180] allowing for easy and structured modelling. However, our approach is largely independent of the chosen formalism and it is enough to assume that the following properties hold:

(P0) there exists a subset $\mathcal{L}_R \subseteq \mathcal{L}$ of *reception locations* corresponding to the information learnt (and stored) by agents from their communication with others;

(P1) LTS's function succ can be partitioned into two successor functions $\mathsf{succ}_{\mathcal{R}}$ and $\mathsf{succ}_{\mathcal{L}}$ that correspond respectively to transitions upon which an agent receives information (and stores it) and to transitions that make progress any agent (including the intruder); (that correspond respectively to the successors that change states or not on the locations from $\mathcal{L}_R$;)

(P2) there is an initial state s_0 and there exists a function slice from state to natural (a measure) such that if $s' \in \mathsf{succ}_{\mathcal{R}}(s)$ then there is no path from s' to any state s'' such that $\mathsf{slice}(s) = \mathsf{slice}(s'')$ and $\mathsf{slice}(s') = \mathsf{slice}(s) + 1$ (it is often call a sweep-line progression [49]);

(P3) there exists a function cpu from state to natural numbers (a hashing) such that for all state s if $s' \in \mathsf{succ}_{\mathcal{L}}(s)$ then $\mathsf{cpu}(s) = \mathsf{cpu}(s')$; mainly, the knowledge of the intruder is not taken into account to compute the hash of a state;

(P4) if $s_1, s_2 \in \mathsf{succ}_{\mathcal{R}}(s)$ and $\mathsf{cpu}(s_1) \neq \mathsf{cpu}(s_2)$ then there is no possible path from s_1 to s_2 and *vice versa.*

More precisely: for all state s and all $s' \in \mathsf{succ}(s)$, if $s'|_{\mathcal{L}_R} = s|_{\mathcal{L}_R}$ then $s' \in \mathsf{succ}_L(s)$, else $s' \in \mathsf{succ}_R(s)$; where $s|_{\mathcal{L}_R}$ denotes the state s whose domain is restricted to the locations in $\mathcal{L}_R$. Intuitively, succ_R corresponds to transitions upon which an agent receives information and stores it.

Here again, concrete models generally make easy to distinguish these two kind of transition.

On concrete models, it is generally easy to distinguish syntactically the transitions that correspond to a message reception in the protocol with information storage. Thus, is it easy to partition succ as above and, for most protocol models, it is also easy to check that the above properties are satisfied. This is the case in particular for the algebra of Petri nets that we have used (see below). Thus, is it easy to partition succ as above. This is the case in particular for the algebra of Petri nets that we have used: the ABCD formalism.

In the following, the presented algorithms compute only S. This is made without loss of generality and it is a trivial extension to compute also T and ℓ, assuming for this purpose that $\mathsf{succ}(s)$ returns tuples $(\ell(s, s'), s', \ell(s'))$. This is usually preferred in order to be able to perform model-checking of temporal logic properties.

We now show how several successive improvement can be introduced. This results in an algorithm that remains quite simple in its expression but actually efficient and relies on a precise use of a consistent set of observations and algorithmic modifications.

2.3.2 Having those structural informations from ABCD models

(a) Finding the appropriate succ functions

Using ABCD, finding the appropriate succ_R and succ_L successors functions is purely syntactic since receptions for succ_R are rules that read the buffer rcv and other rules for succ_L are all intern rules of agents and all the rules of the attacker. As ABCD expressions are transform to Petri nets, finding locations $s|_{\mathcal{L}_R}$ is also easy. It suffices to check buffers that involve in succ_R. Thus, the properties P1, P2 and P3 can be automatically deduce using our methodology (using ABCD for checking secure protocols). To enforce P4, we just need to ensure that no data received and stored by an agent is ever forgotten, which can be checked syntactically as well by allowing only + and ? accesses on agent's buffers. Indeed, since hashing is based on exactly these buffers content, two states differently hashed must differ on these buffers and thus necessary lead to disjoint executions.

(b) Drawbacks and advantages

One of the advantages of this design is that the attacker model is partially parameterized by the knowledge it is initially given. By giving it only public information (agent identifiers, public keys, *etc.*), it behaves as a malicious agent external to the system. We can also however model the attacker as one of the nodes themselves by simply giving it the private information it requires to identify itself as one, or more specifically everything it needs to "play" as a legitimate agent in some execution trace in the state graph. We could similarly model compromised keys, file identifiers, or any other private information.

For secure protocols, our approach can in some ways be seen as straddling between more temporal logical approaches such as CTLK and pure process algebras such as the asynchronous π-calculus. ABCD offered the "best of both worlds" for our particular problem: its process algebra syntax allows models to be easily defined (compared to temporal logics) and their properties to be checked directly state space. On the other hand, the structured nature of its Petri net semantics allowed us to verify systems with non-deterministic choice and iteration - both essential to constructing an accurate model of some protocols - which often lead to intractable (or worse) model checking problems in pure process algebras.

Drawbacks are the lack of a friendly tool to model the protocols (an algebra is a little hard to read) and mainly that we currently checks only finite sessions.

2.3.3 Increasing local computation time

Using Algorithm 2.2, function cpu distributes evenly the states over the processors. However, each super-step is likely to compute very few states during each super-step because only too few computed successors are locally owned. This also results in a bad balance of the time spent in computation with respect to the time spent in communication. If more states can be computed locally, this balance improves but also the total communication time decreases because more states are computed during each call to function successor.

To achieve this result, we consider a peculiarity of the models we are analysing. The learning phase (2) of the attacker is computationally expensive, in particular when a message can be actually decomposed, which leads to recompose a lot of new messages. Among the many forged messages, only a (usually) small proportion are accepted for a reception by agents. Each such reception gives rise to a new state.

This whole process can be kept local to the processor and so without cross-transitions. To do so, we need to design a new partition function cpu_R such that, for all states s_1 and s_2, if $s_1|_{\mathcal{L}_R} = s_2|_{\mathcal{L}_R}$ then cpu_R(s_1)=cpu_R(s_2). For instance, this can be obtained by computing a hash (modulo the number of processors) using only the locations from $\mathcal{L}_R$.

On this basis, function successor can be changed as shown in Algorithm 2.3.

```
1  def successor(known, todo) is
2    tosend ←∅
3    while todo ≠ ∅
4      pick s from todo
5      known ←known ∪ {s}
6      for s′ in succ_L(s) \ known
7        todo ←todo ∪ {s′}
8      for s′ in succ_R(s) \ known
9        tosend ←tosend ∪ {(cpu_r(s′),s′)}
10   return tosend
```

Figure 2.3. An exploration to improve local computation.

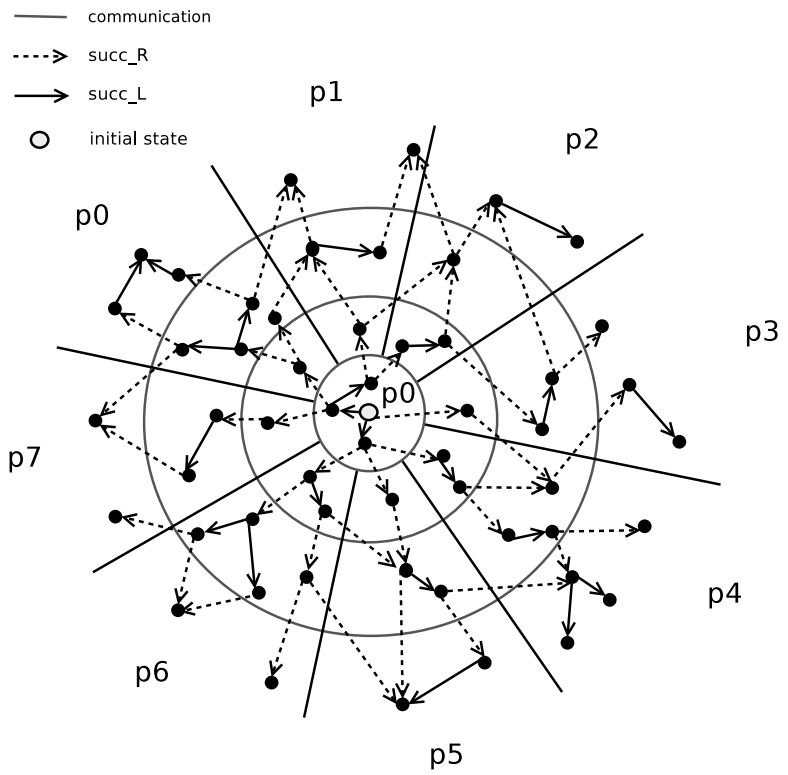


Figure 2.4. Generation of the state space (with local computation) across the processors.

With respect to Algorithm 2.2, this one splits the for loop, avoiding calls to cpu_R when they are not required. This may yield a performance improvement, both because cpu_R is likely to be faster than cpu and because we only call it when necessary. But the main benefits in the use of cpu_R instead of cpu is to generate less cross transitions since less states are need to be sent. Finally, notice that, on some states, cpu_R may return the number of the local processor, in which case the computation of the successors for such states will occur in the next super-step. We show now on how this can be exploited.

Figure 2.4 illustrates the generation of state space across the processors by Algorithm 2.3. Initially only processor P0 performs the computation. The cross transitions correspond to the successors succ_R, *i.e.* transitions upon which an agent receives information, other transitions being local, particularly those corresponding to the learning phase of the attacker.

Figure 2.5 expresses the same algorithm (2.3). In this diagram the distribution per processor is not represented, but the global progression of the computation of the state space is showed by slice. An explored state is local to a slice (more precisely to a class as we shall see), *i.e.* it do not be explored in an other slice.

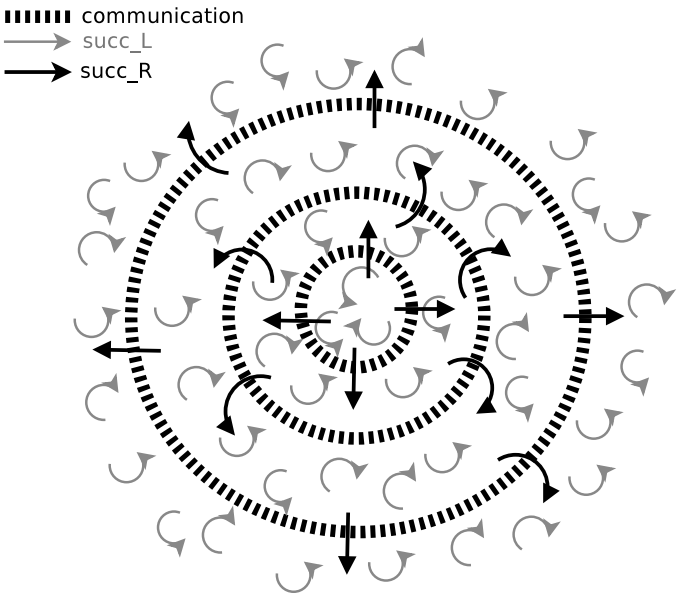


Figure 2.5. Generation of the state space with local computation.

```
1 def exchange(tosend, known) is
2   dump(known)
3   return BSP_EXCHANGE(tosend)
```

Figure 2.6. Sweep-line implementation — the rest is as in Algorithm 2.3.

2.3.4 Decreasing local storage: sweep-line reduction

One can observe that the structure of the computation is now matching closely the structure of the protocol execution: each super-step computes the executions of the protocol until a message is received. As a consequence, from the states exchanged at the end of a super-step, it is not possible to reach states computed in any previous super-step. Indeed, the protocol progression matches the super-steps succession.

This kind of progression in a model execution is the basis of the *sweep-line* method [49] that aims at reducing the memory footstep of a state space computation by exploring states in an order compatible with progression. It thus becomes possible to regularly dump from the main memory all the states that cannot be reached anymore. Enforcing such an exploration order is usually made by defining on states a measure of progression. In our case, such a measure is not needed because of the match between the protocol progression and the super-steps succession. So we can apply the sweep-line method by making a simple modification of the exploration algorithm, as shown in Algorithm 2.6.

Statement dump(known) resets known to an empty set, possibly saving its content to disk if this is desirable. The rest of function exchange is simplified accordingly.

Figure 2.7 represents the progress of the computation of the state space according to the Algorithm 2.6 by explaining the sweep-line (shaded area) of the states already explored in the previous slice. They are indeed “swept” because they do not affect the ongoing computation.

2.3.5 Balancing the computations

As one can see in the future benchmarks in Section 4.3, Algorithm 2.6 (and in the same manner Algorithm 2.3) can introduce a bad balance of the computations due to a lack of information when hashing only on $\mathcal{L}_R$. The final optimisation step aims thus at balancing the workload.

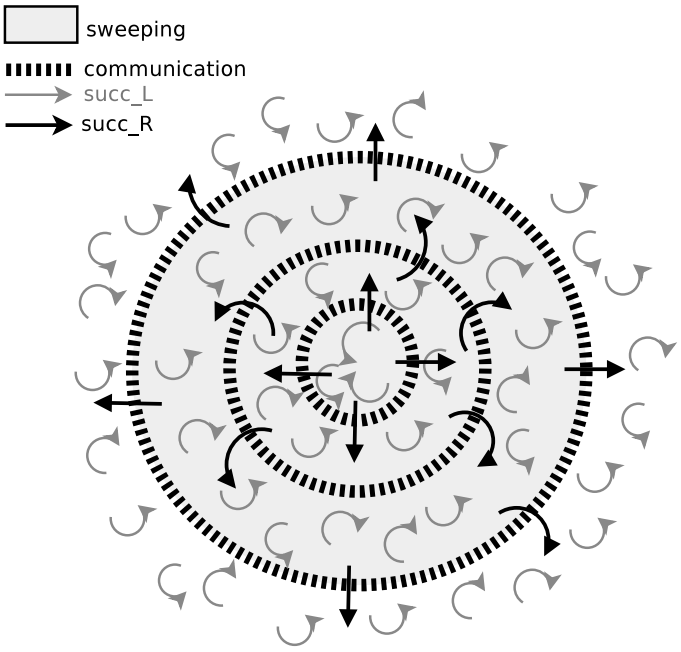


Figure 2.7. Sweep-line generation of the state space.

To do so, we exploit the observation that, for all the protocols that we have studied so far, the number of computed states during a super-step is usually closely related to the number of states received at the beginning of the super-step. So, before to exchange the states themselves, we can first exchange information about how many state each processor has to send and how they will be spread onto the other processors. Using this information, we can anticipate and compensate balancing problems.

To compute the balancing information, we use a new partition function cpu_B that is equivalent to cpu_R without modulo, *i.e.* we have cpu_R(s) = cpu_B(s) **mod** p, where p is the number of processors. This function defines classes of states for which cpu_B returns the same value. We compute a histogram of these classes on each processor, which summarises how cpu_R would dispatch the states. This information is then globally exchanged, yielding a global histogram that is exploited to compute on each processor a better dispatching of the states it has to send. This is made by placing the classes according to a simple heuristic for the bin packing problem: the largest class is placed onto the less charged processor, which is repeated until all the classes have been placed. It is worth noting that this placement is computed with respect to the global histogram, but then, each processor dispatches only the states it actually holds, using this global placement. Moreover, if several processors compute a same state, these identical states will be in the same class and so every processor that holds such states will send them to the same target. So there is no possibility of duplicated computation because of dynamic states remapping.

These operations are detailed in Algorithm 2.8 where variables histoL and histoG store respectively the local and global histograms, and function BinPack implements the dispatching method described above. In function balance, $\sharp X$ denotes the cardinality of set X. Function BSP_MULTICAST is used so that each processor sends its local histogram to every processor and receives in turn their histograms, allowing to build the global one. Like any BSP communication primitive it involves a synchronisation barrier.

It may be remarked that the global histogram is not fully accurate since several processors may have a same state to be sent. Nor the computed dispatching is optimal since we do not want to solve a NP-hard bin packing problem. But, as shown in our benchmarks below, the result is yet fully satisfactory.

Finally, it is worth noting that if a state found in a previous super-step may be computed again,

```
1 def exchange(tosend, known) is
2   dump(known)
3   return BSP_EXCHANGE(Balance(tosend))
```

```
1 def balance(tosend) is
2   histoL ←{ (i, ♯{ (i,s) ∈ tosend}) }
3   compute histoG from BSP_MULTICAST(histoL)
4   return BinPack(tosend, histoG)
```

Figure 2.8. Balancing strategy — the rest is as in Algorithm 2.6, using cpu_B instead of cpu_R.

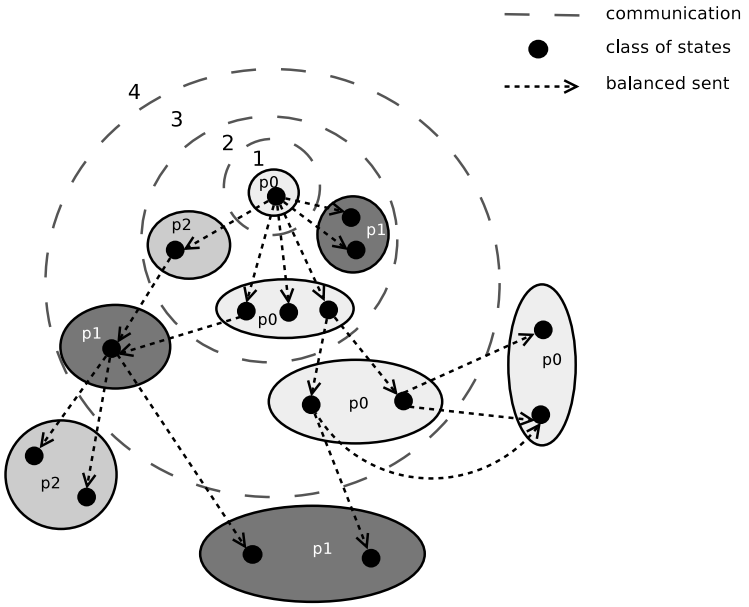


Figure 2.9. Generation of the state space with balancing strategy.

it would be necessary to known which processor owns it: this could not be obtained efficiently when dynamic remapping is used. But that could not happen thanks to the exploration order enforced in Section 2.3.3 and discussed in Section 2.3.4. Our dynamic states remapping is thus correct because states classes match the locality of computation.

Figure 2.9 shows the computation of the state space by the algorithm 2.8 following our balancing strategy. Classes of state are distributed over the processors taking into account of their weight. A state explored in a class can not be in another class.

2.4 Formal explanations of the LTS hypothesis

2.4.1 General assumptions

Here we give some explanations concerning the accuracy of our methods. Why our improvements hold on the protocol modeled, a very wide range of protocol models, we consider. We write more precisely and more formally the hypothesis to be verified by our algorithms support. These assumptions hold for a very wide range of protocol models and are the basis of improvements that we propose. Then we discuss briefly the question of the adaptability of our algorithms to other types of protocols and in particular the branching protocols, this exceeding the borders of the work proposed in this thesis.

In a first time, we recall more formally, the definitions of state space and the transition relation on the states.

We consider the finite set of the states $S \stackrel{\text{df}}{=} \{s_1, ..., s_{\alpha<\infty}\}$, and the finite set of the transitions of the underlying model $T \stackrel{\text{df}}{=} \{t_1, ..., t_{\beta<\infty}\}$. The set of the edges is $E \subseteq S \times T \times S$. The initial state is s_0.

For each transition $t \in T$, the firing relation expressing the edges is denoted by $\rightarrow_t$ and defined with respect of the edges, *i.e.* $s \rightarrow_t s'$ if $(s, t, s') \in E$.

We recall the definition of the notion of successor in relation to this states space.

Definition 5 (*Successors of a state*).
For each state s, $succ(s)$ denotes the set of successors according to the edges:

$$succ(s) \stackrel{\text{df}}{=} \{s' | \exists t \in T / s \rightarrow_t s'\}$$

If $s' \in succ(s)$, we note $s \rightarrow s'$.

We consider also the successor of a certain subset $X \subset T$ of the transitions:

$$succ_X(s) \stackrel{\text{df}}{=} \{s' | \exists t \in X / s \rightarrow_t s'\}$$

We use a functional notation for the transitions for reasons of ease of writing.

Definition 6 (*Functional notation for the transitions*).
We use the notation $t(s) = s'$ to express $s \rightarrow_t s'$ and we extend this notation to the sets. Considering some set of transitions X and set of states Y_1, Y_2, we note $X(Y_1) \subseteq Y_2$ if $\forall t \in X, \forall s \in Y_1, t(s) \subseteq Y_2$. The transitions of X acting on Y_1 goes in Y_2.

Here we summarize the conditions that must be provided by our models. Our models check the following conditions:

Definition 7 (*General assumptions*).

1. *(Combination of local states) All the states of our models are the combination of local states. In this way, the set of states S can be seen as the subset of the product of local states sets:*

   $$S \subseteq S_1 \times ... \times S_{\alpha<\infty}$$

   Each local set of states being disjoint from another: $i \neq j \Rightarrow S_i \cap S_j = \emptyset$. For each $s \in S$ and some $i \in \{1, ..., \alpha\}$, s_i stands for the local component of s which belong to the state component S_i.

2. *There exists a partition of the set of the local states: there exits nonempty sets S_{local} and S_{rcv} ensuring $S \subseteq S_1 \times ... \times S_\alpha = S_{local} \uplus S_{rcv}$. Without loss of generalites, we note $S_{local} = S_1 \times ... \times S_a$, $S_{rcv} = S_{a+1} \times ... \times S_k$ and $s|_{local}$ (resp. $s|_{rcv}$) stands for the local component of s,* i.e. *the component of s in relation with S_{local} (resp. S_{rcv}).*

3. *There exists a partition of the transitions: $T = T_{local} \uplus T_{rcv}$ where $T_{local} = \{t_1, ..., t_b\}$ and $T_{rcv} = \{t_{b+1}, ..., t_\beta\}$.*

4. *The local transitions are invariant on the reception component of the states,* i.e. *T_{local} is invariant on S_{rcv}:*

   $$\forall t \in T_{local}, s \rightarrow_t s' \iff s|_{rcv} = s'|_{rcv}$$

5. *There exists a partition of S_{rcv}:*

   $$S \subseteq S_{rcv0} \times ... \times S_{rcv\gamma}$$

   Each S_{rcvi} marks the progression of each agent i in the protocol.

6. *We assume that each* $S_{rcvi} = \{s_{i1}, ..., s_{i\delta}\}$ *is totally ordered by* $\prec_i$. *Without loss of generality, one supposes forall i* $s_{i1} \prec_i ... \prec_i s_{i\delta}$.

7. T_{rcv} *changes necessarily a single component of* S_{rcv}. *Formally,*

$$\forall t \in T_{rcv}, \forall s, s' \in S, s \rightarrow_t s' \Rightarrow$$

$$\exists! i \in \{0, ..., \gamma\} / s|_{rcvi} \neq s'|_{rcvi} \text{ and } \forall j \neq i, s|_{rcvj} = s'|_{rcvj}$$

Moreover, we consider (hypothesis of strict progression of agents): $s|_{rcvi} \prec_i s'|_{rcvi}$.

8. *A set of function* $\{progress_0, ..., progress_\gamma\}$ *such as:*
   - $progress_i : S_{rcvi} \rightarrow \mathbb{N}$;
   - $\forall i \in \{0, ..., \gamma\}, progress_i(s_{i1}) = 0$;
   - *and* $progress_i(s_{i(j+1)}) = progress_i(s_{ij}) + 1$.

9. *We consider that the firing of a transition t from* T_{rcv} *"follows" the strict order of a certain component reception, we consider* S_{rcvi}:

$$s \rightarrow_t s' \Rightarrow s \text{ and } s|_{rcvi} = s_{ij} \text{ then}$$

$$s|_{rcvi} = s_{i(j+1)}$$

10. *We note* $progress : S \rightarrow \mathbb{N}$ *defined forall s by*

$$progress(s) \stackrel{\text{df}}{=} \sum_{i \in \{0,...,\gamma\}} progress_i(s|_{receptioni})$$

The items 1 and 2 implie the property (P0) of our generic protocols model.
By item items 3, 4 and 7 we give (P1).
We consier our cpu function acts on $S_{reception}$. By items 4 and 7 it follows (P3) and (P4).
We consider the function *progress* as the function slice of the part 2.3.1. It follows by items 7, 8, 9 and 10 the property (P2) and we add slice(s_0)=0.

2.4.2 Slices

The *progress mesure* is the fundamental concept underlying the sweep-line method. The key property of a progress measure is that for a given state s, all states reachable from s have a progress measure which is greater or equal than the progress measure of s. A progress measure makes it possible to delete certain states on-the-fly during state space generation, since it ensures that these states can never be reached again. We recall the definition from [49].

Definition 8 (*Progress mesure [49]*).
A progress mesure is a tuple $(O, \sqsubseteq, \varphi)$ *such that* $(O, \sqsubseteq)$ *is a partial order and* $\varphi : S \rightarrow O$ *is a mapping from states into O satisfying:*

$$\forall s, s' \in S, s \rightarrow^* s' \Rightarrow \varphi(s) \sqsubseteq \varphi(s')$$

Our previous assumptions provide a progress mesure ($\mathbb{N}, \leq$,slice) for the states of the system:

$$\forall s, s' \in S, s \rightarrow^* s' \Rightarrow \text{ slice}(s) \leq \text{ slice}(s')$$

Our algorithms send only the states fired by reception transition and involve:

$$\forall s, s' \in S, \forall t \in T_{rcv}, s \rightarrow^t s' \Rightarrow \text{ slice}(s') = \text{ slice}(s) + 1$$

It is correct to dump the states computed before the new reception states (corresponding to the receptions by reception transitions), the strict progression by reception transitions ensuring that we will do not find them in the future.
We obtain, thus, an exploration by slice, each slice defined as follows:

Definition 9 (*Slices*).
We define the slices: $slice_1, ..., slice_{\varpi<\infty}$ of S by:

$$slice_i \stackrel{\text{df}}{=} \{s | \mathsf{slice}(s) = i\}.$$

We note that one can know *a priori* the number of super step involved in the generation of the state space making easiest the problem of the termination of the algorithm (not used in practice because of the simplicity of writing and understanding provided by the BSP model). Indeed, each process-agent consist as a certain number of strict progression point in the protocol represented by some places of reception, $\mathcal{L}_R$, (*i.e.* taking value in S_{rcv}). The number of these places gives the number of super-steps.

We followed the approach proposed by [149] to distribute the states accross the machines by selecting a certain processus: the *designated* process with some relevance unstead of hashing on the complete state (*i.e.* all the components of the state). More exactly the more general approach advocated by [142] by selecting some relevant places.

2.4.3 Receptions and classes

Here we make a clear choice concerning the *designated* places: the reception location $\mathcal{L}_R$ corresponding to the progress steps of each agent (and not the attacker) and thus to the advance of the protocol, making behavior of the attacker merely local. The set of the local transitions (T_{local}) maintaining this set we do not need to test the localization (by cpu) of states obtained by T_{local}.

The computation of the state space, during the super steps can be thus be seen by a class concept with respect to the reception states.

We define the states of reception as those received by the machine:

Definition 10 (*Reception state*).
A state $s \in S$ is a reception state if it is the initial state $s = s_0$ or if it is fired by a reception transition, formally: $\exists s' \in S / s' \rightarrow_t s \wedge \in T_{rcv}$.

We can differentiate these reception states, by distinction of the reception component. We group the reception states relatively to a homogeneous component of reception. These states generate a set of states by local transitions, all state forming a *reception class.*

Definition 11 (*Reception class*).
A reception class $\mathcal{C}$ of S is defined from a set of reception states $\{s_1, ..., s_k\}$ by:

- $s_1|_{rcv} = ... = s_k|_{rcv}$;
- $\{s_1, ..., s_k\} \subseteq \mathcal{C}$;
- $(s' \in \mathcal{C}$ *and* $s'' \in succ_{local}(s')) \Rightarrow s'' \in \mathcal{C}$.

By items 4, 7 of general assumptions and the definition succ, it follows that the states of a same reception class have the same reception component, and that two different classes do not share no states. This provides better locality in the computation.

We note that only the reception component is used for the distribution on the machines, this corresponding to the designation of the places of receptions of the model for the distribution function cpu. The conservation of this component following local transitions provides a non-duplication of the states accross the machines.

The relation $\mathcal{R}$ notes the fact that two states belong to the same reception class. $\mathcal{R}$ is an equivalence relation. It follows by the point 4 of the general assumptions that:

$$\forall s, s' \in S, s\mathcal{R}s' \Rightarrow s|_{rcv} = s'|_{rcv}$$

where $|_{rcv}$ denotes the local component relatively to the reception places to the state to which it applies.

The state space quotiented by $\mathcal{R}$, corresponds to a dag whose each node is a reception class. It's this dag that we distribute on the machines, the links between class being the cross transitions. The width of this dag is the main argument for the parallelisation, the homogeneous distribution of each slices ensuring a better balance which is the basis for our last improvement.

If $S_{\mathcal{R}}$ stands for the set of reception class and $\rightarrow^*_{rcv}$ for the successor relation by T_{rcv} and $\varphi(s) = \mathcal{R}(s)$ *i.e.* φ maps a state into the reception class component to which it belongs, then $(S_{\mathcal{R}}, \rightarrow^*_{rcv}, \varphi)$ is the progress mesure for our exploration. Since each SCC is contained in a single reception class (but two SCCs can be found in the same reception class), the graph of reception classes contains (or abstract) the graph of SCCs. If S_{SCC} the set of strongly connected components of the state space, $\rightarrow^*_{SCC}$ the reachability relation between the strongly connected SCC components and $\varphi(s) = SCC(s)$, *i.e.* φ maps a state into the strongly connected component to which it belongs, the progress mesure corresponding to the SCCs is $(S_{SCC}, \rightarrow^*_{SCC}, \varphi)$ and offers maximal reduction for the sweep-line method [49].

The question of the branching protocols and of the protocol with loop can be asked about our methods but outs the frameworks of this thesis and will be the subject of future works.

2.4.4 Termination of the algorithms

The set of the states S is defined inductively as the smallest set E such as $s_0 \in E$ and $s \in E \Rightarrow \text{succ}(s) \subseteq E$.

By assumption, each successor set of any state is finite: $\forall s \in S, \exists n \in \mathbb{N}, |\text{succ}(s)| = n$.

If we index the sets *received* and *tosend* by IDs of the processors whose they belong :

$$received_i = \bigcup_{j \in \{0,\ldots,\mathbf{p}-1\}, j \neq i} tosend_j[i]$$

Variable *total* can be found if each machine sends furthermore, couple $(target, |tosend[target]|)$ that informs each machine of an eventual termination, *i.e.* if the second component of each pair is null. Note that this method may provide false-positive (only for the naive algorithm), *i.e.* we don't stop the algorithm while we should, in this case sent states are already known, but then the next super step will be the last. Formally,

$$total = \sum_{i=0}^{\mathbf{p}-1} |received_i| = 0 \Rightarrow \text{Termination of BSP exploration.}$$

2.4.5 Balance considerations

In a certain way, this algorithm considers an arbitrary number of parallel machines or more exactly more machine than there are really (by considering a hashing function without modulo reduction to the number of processors). In pursuing these considerations, before the actual sendings of states, potential sendings and potential receptions is done on potential machines. Locally, each (real) machine counts the number of states which it will send to the potential machines; this is gathered into vectors *histo* such as $histo[j]$ indicates that the current real machine sends to the potential machine j quantity $histo[j]$ of states. A first barrier of communication allows the exchange vectors *histo*, allowing notwithstanding a possible redundancy of receipt, to predict the amount that each potential machine receives. Each real machine knows the actual amount that each potential machine receives. It suffices then to simply associate at each potential machine a real machine to know the number of actual receiption for each (real) machine. Find a such association in order to resolve an ideal balance amounts to solving the NP-complete Knapsack Problem. We find a such association by a simple but efficacious heuristic. First of all, we collect informations of all table $histo_i, i \in \{0, ..., p-1\}$ in a single table $histo_{total}$:

$$histo_{total} \leftarrow \sum_{i=1}^{p} histo_i[k], \text{forall each potential machine } k$$

Next, we consider the knapsacks, that is to say the **p** "real" processors, which we associate initially a zero weight:

$$knapsack[i] \leftarrow 0, \forall i \in \{0, ..., \mathbf{p} - 1\}$$

We consider at least a table, let *table*, using finally as localiaztion balanced function: initially, $table[k] \leftarrow 0$ forall potential machine k; each potential machine corresponding to a hashage unreduced modulo **p**. To balance uniformly the states over the (real) machines, we use in the run the loop below after, assuming a function **max** (reps. **min**) which return the index of the greater (resp. lower) element of an array of integers:

1: **while** $k \leftarrow \mathbf{max}\{histo_{total}\} \neq 0$ **do**
2: $m \leftarrow \mathbf{min}\{knapsack\}$
3: $knapsack_m \leftarrow knapsack_m + histo_{total}[k]$
4: $table[k] \leftarrow m$
5: **end while**

The idea of the algorithm is as follows: in every loop execution, we choose the heaviest hashage k, *i.e.* the potential machine wich received the more of states, and place this in the lightest *knapsack*: m, *i.e.* the real machine which received, for the time being, the least of states. This heuristic gives good results in practice. Note, nevertheless that we ignore possible redundancy in communication. Practice, that motivated this balancencement algorithm, shows that with increasing of processors the redundancy of communication decreases whereas the ratio (size of received states / size of computed states) is more uniform.

3 Model checking

This chapter extends the work of [105].

Contents

3.1 Tarjan 57
3.1.1 Recursive Tarjan algorithm 58
3.1.2 Iterative Tarjan algorithm 59
3.2 Temporal logics LTL and CTL* 60
3.2.1 Notations 61
3.2.2 CTL* syntax and semantics 61
3.2.3 Proof-structures for verifying a LTL formula 63
3.3 LTL checking 65
3.3.1 Sequential recursive algorithm for LTL 65
3.3.2 Sequential iterative algorithm for LTL 67
3.3.3 Parallel algorithm for LTL 68
3.4 CTL* checking 70
3.4.1 Sequential algorithms for CTL* 73
3.4.2 Naive parallel algorithm for CTL* 78
3.4.3 Parallel algorithm for CTL* 85

In what follow, we recall the well known Tarjan algorithm which is the underlying algorithm of the work on a on-the-fly approach used by [28] for CTL* model checking. But the peculiarity of our work concerns parallel computation: our main contributions in the following sections are thus essentially the parallel adaptation of the algorithms found in [28] for the case of security protocols.

In a first time, we recall the Tarjan algorithm for finding the SCCs components of a directed graph. Then we recall the LTL and CTL* logics and give and discuss our relatives parallel algorithms.

3.1 Tarjan

Tarjan's well known algorithm (named from its author, Robert Tarjan [194]) is a graph algorithm for finding the strongly connected components of a graph which requires one depth-first search. A strongly connected components or SCC of a directed graph is a maximal component in which every vertex can be reached from each other — maximal in the sense that if we add any other node we breaks the mutual reachability property

Notice that in the following, we call succ the successor function in the LTS.

```
1  def dfs(σ) is
2    σ.low ←σ.dfsn ←dfn
3    dfn ←dfn+1
4    stack.push(σ)
5    σ.V ←True
6    σ.instack ←True
7    for σ′ in succ(σ)
8      if σ′.V
9        if σ′.instack
10         σ.low ←min(σ.low,σ′.low,σ′.dfsn)
11     else
12       dfs(σ′)
13       σ.low ←min(σ.low,σ′.low)
14   if σ.low = σ.dfsn
15     var top ←⊥
16     ⟨ new_scc ←∅ ⟩
17     while top ≠ σ
18       top ←stack.pop()
19       ⟨ new_scc.add(top) ⟩
20       top.instack ←False
21     ⟨ scc.add(new_scc) ⟩
```

```
1  def tarjan(σ0) is
2    var dfn ←0
3    var stack ←ϵ
4    ⟨ var scc ←∅ ⟩
5    def dfs(σ) is (...)
6    dfs(σ0)
7    ⟨ return scc ⟩
```

Figure 3.1. Recursive Tarjan's Algorithm for SCCs computation.

3.1.1 Recursive Tarjan algorithm

In Figure 3.1 one can see the classical (recursive) Tarjan algorithm. At the termination of the algorithm, the set scc will contains the set of all SCCs of the input graph — for convenience, only the initial state σ_0 is given as argument of procedure tarjan. The algorithm consist of a depth-first exploration by recursive calls, each state possessing two fields: .dfsn and .low expressing respectively the depth-first search number (by incrementation of the dfn variable) and the smallest depth-first search number of the state that is reachable from the considering state. The detection of the belonging of a state to a SCC is made by the belonging test of a successor to the global stack. A SCC is found at a certain point if at the end of some recursive call, the field .low coincides with the field .dfsn.

The Tarjan algorithm has been design for calculating all the SCCs of a given graph. But in the following (verification of a logical formulae), we will limit ourselves to only finding one SCC. In the algorithm of Figure 3.1, we thus put into chevrons what is dedicated to compute all the SCCs and which not be useful in the next sections.

For a more intuitive understanding of this algorithm, we trace its execution for a simple example given as a LTS shown in Figure. 3.2:

```
1  tarjan(A)
2    dfn=0, stack=ϵ
3    dfs(A)
4      A.low = A.dfsn = 0
5      dfn = 1
6      stack = [A]
7      dfs(B)
8        B.low = B.dfsn = 1
9        dfn = 2
10       stack = [A,B]
11       dfs(C)
12         C.low = C.dfsn = 2
13         dfn = 3
14         stack = [A,B,C]
15         C.low = 1
16       dfs(D)
17         D.low = D.dfsn = 3
18         dfn = 4
19         stack = [A,B,C,D]
20         dfs(E)
21           E.low = E.dfsn = 4
22           dfn = 5
23           stack = [A,B,C,D,E]
24           E.low = 0
25           dfs(F)
26             F.low = F.dfsn = 5
27             dfn = 6
28             stack = [A,B,C,D,E,F]
29             dfs(G)
30               G.low = G.dfsn = 6
31               dfn = 7
32               stack = [A,B,C,D,E,F,G]
33               dfs(H)
34                 H.low = H.dfsn = 7
```

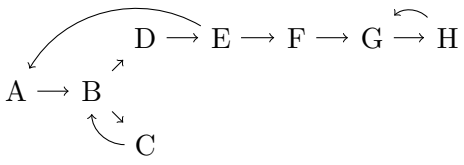


Figure 3.2. LTS example for the Tarjan's Algorithm.

```
1  def call(σ) is
2    σ.low ←σ.dfsn ←dfn
3    dfn ←dfn +1
4    stack.push(σ)
5    σ.V ←True
6    σ.instack ←True
7    σ.children ←succ(σ)
8    loop(σ)

1  def loop(σ) is
2    while σ.children ≠ ∅
3      σ′ ←σ.children.pick()
4      if σ′.V
5        if σ′.instack
6          σ.low ←min(σ.low,σ′.low,σ′.dfsn)
7      else
8        todo.push(σ′)
9        σ′.parent ←σ
10       return {stop the procedure}
11   if σ.low = σ.dfsn
12     var top ←⊥
13     ⟨ new_scc ←∅ ⟩
14     while top ≠ σ
15       top ←stack.pop()
16       ⟨ new_scc.add(top) ⟩
17       top.instack ←False
18     ⟨ scc.add(new_scc) ⟩
19   ret(σ)

1  def up(σ,σ′) is
2    σ.low ←min(σ.low,σ′.low,σ′.dfsn)
3    loop(σ)

1  def ret(σ) is
2    if σ.parent ≠ ⊥
3      up(σ.parent,σ)

1  def tarjan(σ0) is
2    var dfn ←0
3    var stack ←ϵ
4    ⟨ var scc ←∅ ⟩
5    var todo ←[σ0]
6    def up(σ,σ′) is (...)
7    def loop(σ) is (...)
8    def call(σ) is (...)
9    def ret(σ) is (...)
10   while todo
11     σ ←todo.pop()
12     call(σ)
13   {return scc}
```

Figure 3.3. Iterative Tarjan algorithm for SCC computation.

```
35                 dfn = 8
36                 stack = [A,B,C,D,E,F,G,H]
37                 H.low = 7
38               new_scc = {G,H}
39               stack = [A,B,C,D,E,F]
40             new_scc = {F}
41                 stack = [A,B,C,D,E]
42           D.low = 0
43         B.low = 0
44       new_scc = {A,B,C,D,E}
45       stack = ϵ
```

3.1.2 Iterative Tarjan algorithm

We now give an iterative and sequential version of the previous algorithm which we will use later for parallel computations — an iterative version is close to our previous parallel algorithm for parallel state-space computation. Instead of the recursive dfs procedure, we use procedures call, loop, up and ret and an additional stack, todo (which contains initially the initial state) to achieve a derecursification of the previous algorithm. This iterative version is presented in Figure 3.3.

Roughly speaking, the difference lies mainly in the call of dfs which is replaced by a break of the procedure loop to resume the exploration by popping the stack todo in which we have placed the next state to explore. The backtracking is done by the procedure ret which restores the control to its parent call, that in turn may possibly resume the exploration of its children.

Note the definition of subroutines in the main procedure tarjan without their body which are

given separately. This notation is used to define the reach of variables and to decompose the algorithm into several routines.

For the same LTS as above, we give the execution trace of this algorithm:

```
1  tarjan(A)
2   dfn = 0
3   stack = ϵ
4   todo = [A]
5
6   −Loop iteration #1−
7   todo = ϵ
8   call(A)
9    A.low = A.dfsn = 0
10   dfn = 1
11   stack = [A]
12   A.children = {B}
13   loop(A)
14    A.children = ∅
15    todo = [B]
16
17  −Loop iteration #2−
18  todo = ϵ
19   call(B)
20    B.low = B.dfsn = 1
21    dfn = 2
22    stack = [A,B]
23    B.children = {C,D}
24    loop(B)
25     B.children = {D}
26     todo = [C]
27
28  −Loop iteration #3−
29  todo = ϵ
30   call(C)
31    C.low = C.dfsn = 2
32    dfn = 3
33    stack = [A,B,C]
34    C.children = {B}
35    loop(C)
36     C.children = ∅
37     C.low = 1
38     ret(C)
39     up(B,C)
40     loop(B)
41      B.children = ∅
42      todo = [D]
43
44  −Loop iteration #4−
45  todo = ϵ
46   call(D)
47    D.low = D.dfsn = 3
48    dfn = 4
49    stack = [A,B,C,D]
50    D.children = {E}
51    loop(D)
52     D.children = ∅
53     todo = [E]
54
55  −Loop iteration #5−
56   todo = ϵ
57   call(E)
58    E.low = E.dfsn = 4
59    dfn = 5
60    stack=[A,B,C,D,E]
61    E.children = {A,F}
62    loop(E)
63     E.children = {F}
64     E.low = 0
65     E.children = ∅
66     todo = [F]
67
68  −Loop iteration #6−
69   todo = ϵ
70   call(F)
71    F.low = F.dfsn = 5
72    dfn = 6
73    stack=[A,B,C,D,E,F]
74    F.children = {G}
75    loop(F)
76     F.children = ∅
77     todo = [G]
78
79  −Loop iteration #7−
80   todo = ϵ
81   call(G)
82    G.low = G.dfsn = 6
83    dfn = 7
84    stack=[A,B,C,D,E,F,G]
85    G.children = {H}
86    loop(G)
87     G.children = ∅
88     todo = [H]
89
90  −Loop iteration #8−
91   todo = ϵ
92   call(H)
93    H.low = H.dfsn = 7
94    dfn = 8
95    stack=[A,B,C,D,E,F,G,H]
96    H.children = {G}
97    loop(H)
98     H.children = ∅
99     H.low = 6
100    ret(H)
101    up(G,H)
102    loop(G)
103     new_scc = {G,H}
104     stack=[A,B,C,D,E,F]
105     ret(G)
106     up(F,G)
107     loop(F)
108      new_scc = {F}
109      stack=[A,B,C,D,E]
110      ret(F)
111      up(E,F)
112      loop(E)
113       ret(E)
114       up(D,E)
115       D.low = 0
116       loop(D)
117        ret(D)
118        up(B,D)
119        B.low = 0
120        loop(B)
121         ret(B)
122         up(A,B)
123         loop(A)
124          new_scc={A,B,C,D,E}
125          stack = ϵ
```

3.2 Temporal logics LTL and CTL*

A temporal logic is used to reasoning about propositions qualified in terms of time *e.g.* "I will be hungry until I eat something". Temporal logics have two kinds of operators: logical operators and modal operators. Logical operators are usual operators as $\wedge$, $\vee$ *etc.* Model operators are used to reason about time as "until", "next-time" *etc.* Quantifiers can also be used to reason about paths *e.g.* "a formula holds on all paths starting from the current state". Temporal logics are thus mainly used in formal verification of systems and programs.

Researchers have devoted considerable attention to the development of automatic techniques, or model-checking procedures, for verifying finite-state systems against specifications expressed using various temporal logics [52].

There is many temporal logics (with different expressivities) but one of them is the most useful and used: CTL* which subsumes the two usefull logics in verification that are LTL (linear-time

temporal logic) and CTL (Computational tree logic). In LTL, one can encode formulae about the future of paths, *e.g.* a condition will eventually be true, a condition will be true until another fact becomes true, *etc.* CTL is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realised. Finally, it is notice that some temporal logics are more expressive than CTL*. It is the case of the μ-calculus and game-semantic logics as ATL* [3]. But their verification is harder and would be considered in future works.

We now formally defined CTL* and LTL and formal properties about this logics.

3.2.1 Notations

We use the following notations.

Definition 12 (*Kripke structure*).
A Kripke structure is a triple (S, ρ, L) where S is a set of states, $\rho \subseteq S \times S$ is the transition relation, and $L \in S \to 2^{\mathcal{A}}$ is the labeling.

Mainly a Kripke structure is a LTS adjunting a labeling function which give verity to given state.

Definition 13 (*Path and related notions*).
Let $M \stackrel{\text{df}}{=} (S, \rho, L)$ be a Kripke structure.

1. *A path in M is a maximal sequence of states $\langle s_0, s_1, ...\rangle$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in \rho$.*
2. *If $x = \langle s_0, s_1, ...\rangle$ is a path in M then $x(i) \stackrel{\text{df}}{=} s_i$ and $x^i \stackrel{\text{df}}{=} \langle s_i, s_{i+1}, ...\rangle$.*
3. *If $s \in S$ then $\Pi_M(s)$ is the set of paths x in M such that $x(0) = s$.*

We will also concentrate on the notion of proof-structure [28] for LTL checking: a collection of top-down proof rules for inferring when a state in a Kripke structure satisfies a LTL formula. In the following, the relation ρ is assumed to be total — thus all paths in M are infinite. This is only convenient for the following algorithms — it is also easy to make ρ total by making final states, that is states without sucesssors, to be successors of themself.

We fix a set $\mathcal{A}$ of atomic propositions, which will be ranged over by $a, a', \cdots$. We sometimes call formulas of the form a or $\neg a$ literals; $\mathcal{L}$ is the set of all literals and will be ranged over by $l, l_1, \cdots$. We use $p, p_1, q, \cdots$ to range over the set of state formulas and $\varphi, \varphi_1, \gamma, \cdots$ to range over the set of path formulas — both formally defined in the following. We also call A and E path quantifiers and the X, U and R constructs path modalities.

3.2.2 CTL* syntax and semantics

Definition 14 (*Syntax of CTL).**
The following BNF-like grammar describes the syntax of CTL.*

$$\begin{array}{rcl} \mathcal{S} & ::= & a \mid \neg a \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid A\mathcal{P} \mid E\mathcal{P} \\ \mathcal{P} & ::= & \mathcal{P} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid X\mathcal{P} \mid \mathcal{P}U\mathcal{P} \mid \mathcal{P}R\mathcal{P} \end{array}$$

We refer to the formulas generated from S as state formulas and those from $\mathcal{P}$ as path formulas. We define the CTL formulas to be the set of state formulas.*

Let us remark that we use a particular construction on the formulas by putting the negation only adjoining to the atoms.

Remark 1 (*Subsets of CTL)**

- *The CTL (Computation Tree Logic) consists of those CTL* formulas in which every*

Informal semantics of the path modality operators:

$$X\varphi : \bullet \to \bullet^{\varphi} \to \bullet \to \bullet \to \bullet \to \cdots$$

$$\varphi_1 U \varphi_2 : \bullet^{\varphi_1} \to \bullet^{\varphi_1} \to \bullet^{\varphi_1} \to \bullet^{\varphi_2} \to \bullet \to \cdots$$

$$\varphi_1 R \varphi_2 : \bullet^{\varphi_2} \to \bullet^{\varphi_2} \to \bullet^{\varphi_2} \to \bullet^{\varphi_2} \to \bullet^{\varphi_2} \to \cdots$$
$$\text{or} \qquad \bullet^{\varphi_2} \to \bullet^{\varphi_2} \to \bullet^{\varphi_2} \to \bullet^{\varphi_1 \wedge \varphi_2} \to \bullet \to \cdots$$

Figure 3.4. Informal semantics of the path modality operators.

occurrence of a path modality is immediately preceded by a path quantifier.

- *The LTL (Linear Temporal Logic) contains CTL* formulas of the form $(A\varphi)$, where the only state subformulas of φ are literals.*

It is usual to have the two following syntaxic sugars: $F\varphi \equiv trueU\varphi$ (finally) and $G\varphi \equiv falseR\varphi$ (globally). LTL and CTL are not disjoint sets of formulas and both are used in model-checking. For example of security properties:

- Fairness is a CTL formula; $AG(recv(c_1, d_2) \Rightarrow EFrecv(c_2, d_1))$ if we suppose two agents c_1 and c_2 that possess digital items d_1 and d_2, respectively, and wish to exchange these items; it asserts that if c_1 receives d_2, then c_2 has always a way to receive d_1.
- The availability of an agent can be a LTL formula; it requiring that all the messages m received by this agent a will be processed eventually; it can be formalised as: $AG(rcvd(a, m) \Rightarrow (F\neg rcvd(a, m)))$

Definition 15 (*Semantic of CTL).**
Let $M = (S, R, L)$ be a Kripke structure with $s \in S$ and x a path in M. Then $\vDash$ is defined inductively as follows:

- $s \vDash a$ *if* $a \in L(s)$ *(recall* $a \in A$*);*
- $s \vDash \neg a$ *if* $s \nvDash a$*;*
- $s \vDash p_1 \wedge p_2$ *if* $s \vDash p_1$ *and* $s \vDash p_2$*;*
- $s \vDash p_1 \vee p_2$ *if* $s \vDash p_1$ *or* $s \vDash p_2$*;*
- $s \vDash A\varphi$ *if for every* $x \in \Pi_M(s)$, $x \vDash \varphi$*;*
- $s \vDash E\varphi$ *if there exists* $x \in \Pi_M(s)$ *such that* $x \vDash \varphi$*;*
- $x \vDash p$ *if* $x(0) \vDash p$ *(recall* p *is a state formula);*
- $x \vDash p_1 \wedge p_2$ *if* $x \vDash p_1$ *and* $x \vDash p_2$*;*
- $x \vDash p_1 \vee p_2$ *if* $x \vDash p_1$ *and* $x \vDash p_2$*;*
- $x \vDash X\varphi$ *if* $x^1 \vDash \varphi$*;*
- $x \vDash \varphi_1 U \varphi_2$ *if there exists* $i \geq 0$ *such that* $x^i \vDash \varphi_2$ *and for all* $j < i, x^j \vdash \varphi_1$*;*
- $x \vDash \varphi_1 R \varphi_2$ *if for all* $i \geq 0$, $x^i \vDash \varphi_2$ *or if there exists* $i \geq 0$ *such that* $x^i \vDash \varphi_1$ *and for every* $j \leq i$, $x^j \vdash \varphi_2$.

The meaning of most of the constructs is straightforwards. A state satisfies $A\varphi$ (resp. $E\varphi$) if every path (resp. some path) emanating from the sate satisfies φ, while a path satisfies a state formula if the initial sate in the path does. X represents a "next-time" operator in the usual sense, while $\varphi_1 U \varphi_2$ holds of a path if φ_1 remains true until φ_2 becomes true. The constructor R may be thought of as a "release" operator: a path satisfies $\varphi_1 R \varphi_2$ if φ_2 remains true until both φ_1 and φ_2 (φ_1 releases the path from the obligations) or φ_2 is always true. Figure 3.4 gives an informal semantics of the path modality operators.

Finally, although we only allow a restricted form of negation in this logic ($\neg$ may only be applied to atomic propositions), we do have the following result:

$$\frac{s \vdash A(\Phi, \varphi)}{true} \text{ if } s \vDash \varphi \quad (R1) \qquad \frac{s \vdash A(\Phi, \varphi)}{s \vdash A(\Phi)} \text{ if } s \nvDash \varphi \quad (R2) \qquad \frac{s \vdash A(\Phi, \varphi_1 \vee \varphi_2)}{s \vdash A(\Phi, \varphi_1, \varphi_2)} \quad (R3)$$

$$\frac{s \vdash A(\Phi, \varphi_1 \wedge \varphi_2)}{s \vdash A(\Phi, \varphi_1) \quad s \vdash A(\Phi, \varphi_2)} \quad (R4)$$

$$\frac{s \vdash A(\Phi, \varphi_1 U \varphi_2)}{s \vdash A(\Phi, \varphi_1, \varphi_2) \quad s \vdash A(\Phi, \varphi_2, X(\varphi_1 U \varphi_2))} \quad (R5)$$

$$\frac{s \vdash A(\Phi, \varphi_1 R \varphi_2)}{s \vdash A(\Phi, \varphi_2) \quad s \vdash A(\Phi, \varphi_1, X(\varphi_1 R \varphi_2))} \quad (R6)$$

$$\frac{s \vdash A(X\varphi_1, ..., X\varphi_n)}{s_1 \vdash A(\varphi_1, ..., \varphi_n) \quad s_m \vdash A(\varphi_1, ..., \varphi_n)} \text{ if } succ(s) = \{s_1, ..., s_m\} \quad (R7)$$

Figure 3.5. Proof rules for LTL checking [28].

Lemma 1 (*Decomposition of a formulae [28]*)

Let $M \stackrel{\text{df}}{=} (S, \rho, L)$ be a Kripke structure.

1. *For any state formula p there is a state formula $neg(p)$ such that for all $s \in S$, $s \vDash neg(p)$ iff $s \nvDash p$.*
2. *For any path formula φ there is a path formula $neg(\varphi)$ such that for all paths x in M, $x \vDash neg(\varphi)$ iff $x \nvDash \varphi$.*

In [28], the authors give an efficient algorithm for model-checking LTL then CTL* formulas. The algorithm is based on a collection of top-down proof rules for inferring when a state in a Kripke structure satisfies a LTL formula. They appear in Figure 3.5 and are "goal-directed" in the sense that the goal of the rule appears above the subgoals. Moreover, they work on assertions of the form $s \vdash A\Phi$ where $s \in S$ and Φ is a set of path formulas.

Semantically, $s \vdash A(\Phi)$ holds if $s \vDash A(\bigvee_{\varphi \in \Phi} \varphi)$. We sometime write $A(\Phi, \varphi_1, \cdots, \varphi_n)$ to represent $A(\Phi \cup \{\varphi_1, \cdots, \varphi_n\})$ and we consider $A(\emptyset) = \emptyset$. If σ is an assertion of the form $s \vdash A\Phi$ then we use $\varphi \in \sigma$ to denote that $\varphi \in \Phi$.

Note that these rules are similar to ones devised in [72] for translating CTL formulas into the modal μ-calculus.

3.2.3 Proof-structures for verifying a LTL formula

This logic permits users to characterize many properties, including safety and liveness.

Semantically, $s \vdash A\Phi$ holds if $s \vDash A(\bigvee_{\varphi \in \Phi} \varphi)$. We write $A(\Phi, \varphi_1, \cdots, \varphi_n)$ to represent a formula of the form $A(\Phi \cup \{\varphi_1, \cdots, \varphi_n\})$. If σ is an assertion of the form $s \vdash A\Phi$, then we use $\varphi \in \sigma$ to denote that $\varphi \in \Phi$. Proof-rules are used to build proof-structures that are defined as follows:

Definition 16 (*Proof structure [28]*).
Let Σ be a set of nodes, $\Sigma' \stackrel{\text{df}}{=} \Sigma \cup true$, $V \subseteq \Sigma'$, $E \subseteq V \times V$ and $\sigma \in V$. Then $\langle V, E \rangle$ is a proof structure for σ if it is a maximal directed graph such that for every $\sigma' \in V$, σ' is reachable from σ, and the set $\{\sigma'' \mid (\sigma', \sigma'') \in E\}$ is the result of applying some rule to σ'.

Intuitively, a proof structure for σ is a direct graph that is intended to represent an (attempted) "proof" of σ. In what follows, we consider such a structure as a directed graph and use traditional graph notations for it. Note that in contrast with traditional definitions of proofs,

proof structures may contain cycles. In order to define when a proof structure represents a valid proof of σ, we use the following notion:

Definition 17 (*Proof structure successful [28]*).
Let $\langle V, E\rangle$ be a proof structure.

- *$\sigma \in V$ is a leaf iff there is no σ' such that $(\sigma, \sigma') \in E$. A leaf σ is successful iff $\sigma \equiv true$.*
- *An infinite path $\pi = \langle \sigma_0, \sigma_1, \cdots \rangle$ in $\langle V, E\rangle$ is successfull iff some assertion σ_i infinitely repeated on π satisfies the following: there exists $\varphi_1 R \varphi_2 \in \sigma_i$ such that for all $j \geq i, \varphi_2 \notin \sigma_j$.*
- *$\langle V, E\rangle$ is partially successful iff every leaf is successful. $\langle V, E\rangle$ is successful iff it is partially successful and each of its infinite paths is successful.*

Roughly speaking, an infinite path is successful if at some point a formula of the form $\varphi_1 R \varphi_2$ is repeatedly "regenerated" by application of rule R6; that is, the right subgoal (and not the left one) of this rule application appears each time on the path. Note that after $\varphi_1 R \varphi_2$ occurs on the path φ_2 should not, because, intuitively, if φ_2 was true then the success of the path would not depend on $\varphi_1 R \varphi_2$, while if it was false then $\varphi_1 R \varphi_2$ would not hold. Note also that if no rule can be applied (*i.e.* $\Phi = \emptyset$) then the proof-structure and thus the formula is unsuccessful. We now have the following result:

Theorem 1 (*Proof-structure and LTL [28]*)

Let M be a Kripke structure with $s \in S$ and $A\varphi$ an LTL formula, and let $\langle V, E\rangle$ be a proof structure for $s \vdash A\{\varphi\}$. Then $s \vDash A\varphi$ iff $\langle V, E\rangle$ is successful.

One consequence of this theorem is that if σ has a successful proof structure, then all proof structures for σ are successful. Thus, in searching for a successfull proof structure for an assertion no backtracking is necessary. It also turns out that the success of a finite proof structure may be determined by looking at its strongly connected components or any accepting cycle. An obvious solution to this problem would be to construct the proof structure for the assertion and then check if the proof structure is successful. Of course, this algorithm is not on-the-fly as it does not check the success of a proof structure until after it is completely built. An efficient algorithm, on the other hand, combines the construction of a proof structure with the process of checking whether the structure is successful. A Tarjan like algorithm was used in [28] but a NDFS [129] one could also be used.

Call a SCC S of $\langle V, E\rangle$ *nontrivial* if there exist (not necessary distinct) $v, v' \in S$ such that there is a path containing a least one edge from v to v'. For any $V' \subseteq V$ we may define the *success* set of V' as follows:

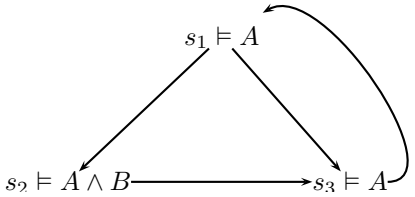
$$Success(V') = \{\varphi_1 R \varphi_2 | \exists \sigma \in V' \text{ such as } \varphi_1 R \varphi_2 \in \sigma \wedge \forall \sigma' \in V' \text{ such as } \varphi_2 \notin \sigma'\}.$$

We say that V' is successful if and only if $Success(V') \neq \emptyset$ we have the following:

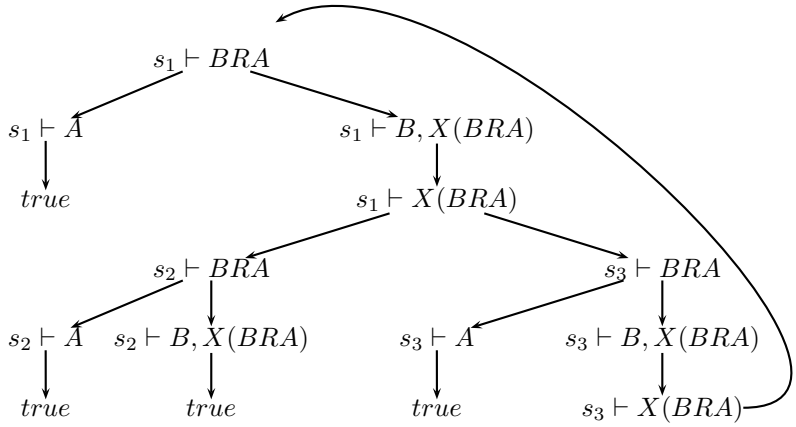
Theorem 2 (*SCC and LTL [28]*)

A partially successful proof structure $\langle V, E\rangle$ is successful if and only if every nontrivial SCC of $\langle V, E\rangle$ is successful.

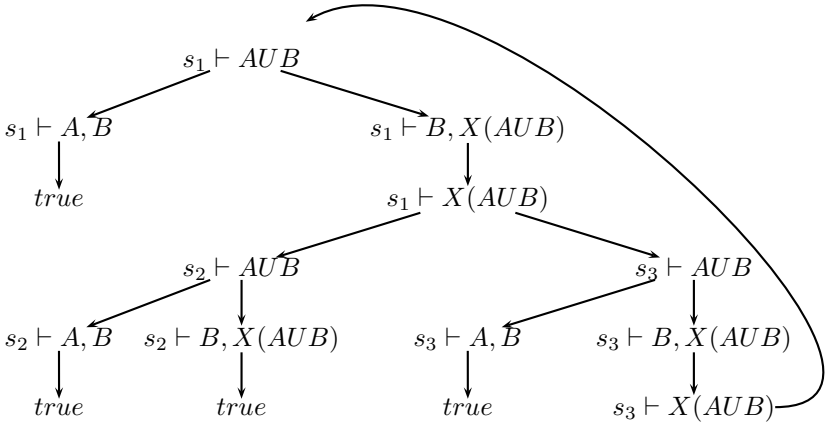
We now give here a simple example of proof-structure on simple Kripke structure:



The atomic proposition A is always true and B is only true for state s_2. Now checking for $s_1 \vdash BRA$ gives use the following successful proof-structure:



and $s_1 \vdash AUB$ give us the following proof-structure:



Once can remark that a proof-structure is not as a natural logical proof: it is a graph. Here, all paths are infinite in the Kripke structure so the proof-structure.

It is notice that there exists different algorithms for on-the-fly checking validity of LTL [52,109] or CTL [201] formula over Kripke structures. One advantage of the approach of [28] is that the CTL* is considered and the authors notes that the complexity are in the same order of magnitude than specialised approaches.

3.3 LTL checking

In this section, we first describe a general sequential algorithm for LTL checking (based on the proof-structures and SCC) and then how parallel it for the problem of security protocols and the state-space generation of the previous chapter.

3.3.1 Sequential recursive algorithm for LTL

Figure 3.6 gives the algorithm for LTL checking of [28]. It is mainly the Tarjan algorithm of Figure 3.1 for finding one sucessful SCC to valide or not the formula: it combines the construction of a proof-structure with the process of checking whether the structure is successful; as soon as it is determined that the partially constructed structure cannot be extended sucessfully, the routine halts the construction of the structure and returns anwser False.

Additional informations is stored in vertices in the structure that enable the detection of unsuccessful SCC. Successors is taken from routine subgoals: it applies the rules of Figure 3.5 and when no subgoal is found an error occurs — unsuccessful proof structure.

```
1  def modchkLTL(σ) is
2    var dfn ←0
3    var stack ←ϵ
4    def init(σ,valid) is (...)
5    def dfs(σ, valid) is (...)
6    def subgoals(σ) is (...)
7    return dfs(σ,∅)

1  def init(σ,valid) is
2    dfn ←dfn+1
3    σ.dfsn ←σ.low ←dfn
4    σ.valid ←{⟨φ1Rφ2,sp⟩ | φ2 ∉ σ
5      ∧(φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)
6      ∧ sp=(sp′ if ⟨φ1Rφ2,sp′⟩ ∈ valid, dfn otherwise}

1  def subgoals(σ) is
2    case σ
3      s ⊢ A(Φ, p) :
4        if (s ⊨ p) then subg←{True}
5        elif Φ = ∅ then subg←∅
6        else subg← A(Φ)
7      s ⊢ A(Φ, φ1 ∨ φ2) :
8        subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
9      s ⊢ A(Φ, φ1 ∧ φ2) :
10       subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
11     s ⊢ A(Φ, φ1Uφ2) :
12       subg←{s ⊢ A(Φ, φ1, φ2),
13       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
14     s ⊢ A(Φ, φ1Rφ2) :
15       subg←{s ⊢ A(Φ, φ2),
16       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
17     s ⊢ A(Xφ1, ..., Xφn) :
18       subg←{s′ ⊢ A(φ1, ...φn) | s′ ∈ succ(s)} (R7)
19   return subg
```

```
1  def dfs(σ, valid) is
2    σ.flag ←True
3    init(σ, valid)
4    σ.V ←True
5    subg ←subgoals(σ)
6    case subg
7      {True} : σ.flag ←True
8      ∅ : σ.flag ←False
9      otherwise :
10       for σ′ ∈ subg while σ.flag
11         elif σ′.V
12           if not σ′.flag
13             σ.flag← False
14           else
15             if σ′.stack
16               σ.low ←min(σ.low,σ′.low)
17               σ.valid ←{⟨φ1Rφ2,sp⟩ ∈ σ.valid | sp≤ σ′.dfsn}
18               if σ.valid = ∅
19                 σ.flag ←False
20         else
21           σ.flag ←dfs(σ′,σ.valid)
22           if σ′.low≤ σ.dfsn
23             σ.low ←min(σ.low,σ′.low)
24             σ.valid ←σ′.valid
25   if σ.dfsn = σ.low
26     var top ←⊥
27     while top ≠ σ
28       top ←stack.pop()
29       if not σ.flag
30         top.flag ←False
31   return σ.flag
```

Figure 3.6. Recursive algorithm for LTL model-checking.

The algorithm uses the following data structures. With each assertion σ we associate three fieds: (1) σ.dfsn, (2) σ.low and (3) σ.valid. The first contains the depth-first search number of σ, while the second records the dept-first search number of the "oldest" ancestor of σ that is reachable from σ — this is used to detect SCC. Finally, the third is a set of pairs of the form $\langle \varphi_1 R \varphi_2$, sp $\rangle$. Intuitively, the formula component of such a pair may be used as evidenve of the success of the SCC that σ might be in, which sp records the "starting point" of the formula, *i.e.* the depth-first number of the assertion in which this occurence of the formula first appeared.

The algorithm also virtually maintains two sets of assertions: V (for visited), which records the assertions that have been encountered so far, and F, which contains a set of assertions that have been determined to be False. To do so, each assertion σ has two adding boolean fields V and flag: we make thus the hypothesis that all computated assertions are in an implicit mapping from the pairs $\langle state, A\Phi \rangle$ (as keys) to fields (this is common in object programming as Python) and when new assertions are computed, these fields are asigned appropriately — V and flag are initially to False.

The heart of the algorithm is routine dfs() which is responsibe for attempting to construct a successul proof structure of its argument assertion σ. Given an assertion σ and a set of formula/number pairs (intuitively, the valid set from σ's parent in the depth-first search tree), the proceure first initializes the dfsn and low fields of σ appropriately, and it assigns to σ's valid field a set of pairs and their "starting points". Note that $\varphi_1 R \varphi_2$ appears in σ.valid and σ's parent then the starting point of the formula is inherited from the parent.

We used a test of membership of assertions in a stack. For this we add another field call stack

to the assertions to have a constant time test.

After pushing σ onto the stack and adding σ to the set V, dfs calls the procedure subgoals which returns the subgoals resulting from the application of a rule to σ. Also if $\sigma' \in \mathsf{subgoal}(\sigma)$ then σ' have its fields assigned appropriately (dfsn, low are the same while V, flag, stack and valid are False or empty set except if the assertion is already in the implicit map.

Procedure dfs then processes the subgoal as follows. First, if the only subgoal has the form True, dfs should return True, while is the set of subgoal is empty, then σ is an unsuccessful leaf, and False should be returned. Finally, suppose the set of subgoals is a nonempty set of assertions, we examine each of there in the following fashion. If subgoal σ' has already been examined (*i.e.* is in V) and found to be False (*i.e.* is in F) then the proof structure cannod be successul, and we terminate the processing in ordrer to return False — we pop all the assertions from the stack and if they are in the same SCC, they are marked to be False. if σ' has not been found False, and if σ' is in the stack (meaning that its SCC is still being constructed), the σ and σ' will be in the same SCC: we reflect this by updating σ.low accordingly. We also update σ.valid by removing formulas whose starting points occur *after* σ'; as we show below, these formulas cannot be used as evidence for the success of the SCC containing σ and σ'.

Note the if σ.valid becomes empty then the proof structure cannot be successful and should return False. On the other hand, if σ' has not been explored the dfs is invoked recursively on σ', and the low and valid fields of σ updated appropriately if σ' is determined to be in the same SCC as σ.

Once the subgoal processing is completed, dfs checks to see whether a new SCC component has been detected; if no, it removes it from the stack.

Note the routine dfs incrementally constructs a graph that, if successfully completed, constitutes a proof structure for the given assertion. The vertex set V is maintened explicitely, with E defined implicitely by: $\langle \sigma, \sigma' \rangle \in E$ if $\sigma' \in subgoals(\sigma)$. Given σ, let $f(\sigma) = \{\varphi | \exists j.$ such as $\langle \varphi, j \rangle \in \sigma.\mathsf{valid}\}$. Also for a set $S \subseteq V$, define $h(S)$ to be the assertion σ with the largest dfsn number. We have the following:

Lemma 2 (*Invariant of the algorithm [28]*)

Procedure dfs *maintains: let $G = \langle V, E \rangle$ be a snapshot of the graph constructed by* dfs *during its execution. Then for every SCC S in G, $f(h(S)) = Success(S)$.*

Now we have:

Theorem 3 (*Correctness of the algorithm [28]*)

When modchkLTL*(σ) terminates, we have that for every $\sigma' \in V$ of the form $s' \vdash A(\Phi')$ then $s' \vDash A(\Phi')$ if and only if $\sigma' \notin F$. Thus,* modchkLTL*($s \vdash A(\varphi)$) return* True *if and only if $s \vDash A(\varphi)$.*

3.3.2 Sequential iterative algorithm for LTL

We now present an iterative version of the previous algorithm. To obtain it, we basically use the same transformations as already used for the derecursification of Tarjan algorithm namely, by replacing the recursive procedure dfs with procedures call_ltl, loop_ltl, ret_ltl and up_ltl and by using an additional stack todo. The dfs procedure is thus roughly the same that his homonym in Tarjan's iterative algorithm but with the difference that it does not use its successors in the graph but its subgoals in the proof graph: it actually performs the same modified and iterative version of the Tarjan algorithm but on the proof graph whose the nodes are assertions, *i.e.* couple of state and logic formula, instead of only states.

The goals remain the same: being able to stop calls to procedure loop_ltl and to resume the exploration by popping stack todo in which we a have placed the next state to explore. The backtracking is made using procedure ret_ltl which restores the control to its parent call_ltl, this one possibly resuming the exploration of its children.

For model-checking LTL formulas, we begin by the procedure modchkLTL which initiates the variables dfn and stack and start the depth exploration by dfs on the assertion argument of modchkLTL. Another difference is the boolean value associated to the assertion expressed by the variable flag. Initially flag is True, and is False either if the set of subgoals of an assertion is empty or if one of these conditions is satisfied:

- one of the subgoals of the assertion is already visited and its flag is False (this case will be possible when we will check CTL* formulas);
- a nontrivial strongly component unsuccessful is found by testing if the set valid is empty or not.

The init procedure corresponds actually to the begining of the dfs procedure in the recursive Tarjan's algorithm in which the initialisation of the field valid is added in the recursive calls.

Figure 3.7 gives the code of the above algorithm. Note that this iterative version of the SCC computation on proof structures will be used in the following. This will facilitate to stop the parallel "depth" researchs for CTL* formulas: using recursive calls, we would have to manage the program's stack which is not easy and depends of the using programming langage — here Python. Using an iterative algorithm, this feature can be easilly added in procedure loop_ltl and ret_ltl.

3.3.3 Parallel algorithm for LTL

As explained in the previous chapter, we use two functions to compute the successors of a state in the Kirpke structure: $\mathsf{succ}_{\mathcal{R}}$ ensures a measure of progression slice that intuitivelly decomposes the Kripke structure into "a list" of slices $[s_0, \ldots, s_n]$ where transitions from states of s_i to states of s_{i+1} come only from $\mathsf{succ}_{\mathcal{R}}$ and there is no possible path from states of s_j to states of s_i for all $i < j$. Moreover, after $\mathsf{succ}_{\mathcal{R}}$ transitions (with different hashing), there is no possible common paths which is due to different knowledge of the agents. In this way, if we assume, as in Chapter 2, a distribution of the Kripke structure across the processors using function cpu_R (for distributed the Kripke structure, we thus naturally extend this function to assertions σ on only the state field; formulas and depth-first numbers are not take into account), then the only possible accepting cycles or SCCs are locals to each processor.

Thus, because proof structures follow the Kripke stucture (rule R7), accepting cycles or SCCs are also only locals. This fact ensures that any sequential algorithm to check cycles or SCCs can be used for the parallel computation.[1] Call this generic algorithm SeqChkLTL which takes an assertion $\sigma \stackrel{\text{df}}{=} s \vdash A\Phi$, a set of assertions to be sent for the next super-step, and (V, E) the sub-part of the proof-graph (a set of assertions as vertices and a set of edges) that has been previously proceed (this sub-part can grow during this computation). Now, in the manner of the computation of the state-space, we can design our BSP algorithm which is mainly an iteration over the independant slices, one slice per super-step and, on each processor, working on independant sub-parts of the slice by calling SeqChkLTL. This parallel algorithm is given in Figure 3.8.

This is a SPMD (Single Program, Multiple Data) algorithm so that each processor executes it. The main function is ParChkLTL, it first calls an initialisation function in which only the one processor that owns the initial state saves it in its todo list. The variable total stores the number of states to be proceeded at the beginning of each super-step; V and E store the proof graph; super_step stores the current super-step number; dfn is used for the SCC algorithm; finally, flag is used to check whether the formula has been proved False (flag set to the violating state) or not (flag=$\perp$).

[1] It is mainly admited that SCC computation is efficient whereas NDFS is memory efficient, and SCC gives smaller traces. Both methods are equivalent for our purpose. In the previous section, following [28], a SCC computation is used and we have the iterative version.

```
1  def modchkLTL(σ) is
2    var dfn ←0
3    var stack ←ϵ
4    var todo ←[σ0]
5    def init(σ,valid) is (...)
6    def loop_ltl(σ) is (...)
7    def up_ltl(σ,σ′) is (...)
8    def ret_ltl(σ) is (...)
9    def subgoals(σ) is (...)
10   while todo ≠ ϵ
11     σ ←todo.pop()
12     call_ltl(σ)
13   return σ0.flag

1  def subgoals(σ) is
2    case σ
3      s ⊢ A(Φ, p) :
4        if (s ⊨ p) then subg←{True}
5        elif Φ = ∅ then subg←∅
6        else subg← A(Φ)
7      s ⊢ A(Φ, φ1 ∨ φ2) :
8        subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
9      s ⊢ A(Φ, φ1 ∧ φ2) :
10       subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
11     s ⊢ A(Φ, φ1Uφ2) :
12       subg←{s ⊢ A(Φ, φ1, φ2),
13       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
14     s ⊢ A(Φ, φ1Rφ2) :
15       subg←{s ⊢ A(Φ, φ2),
16       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
17     s ⊢ A(Xφ1, ..., Xφn) :
18       subg←{s′ ⊢ A(φ1, ...φn) | s′ ∈ succ(s)} (R7)
19   return subg

1  def init(σ,valid) is
2    dfn ←dfn+1
3    σ.dfsn ←σ.low ←dfn
4    σ.valid ←{⟨φ1Rφ2,sp⟩ | φ2 ∉ σ
5      ∧(φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)
6      ∧ sp=(sp′ if ⟨φ1Rφ2,sp′⟩ ∈ valid else dfn)

1  def call_ltl(σ) is
2    # init
3    if σ.parent = ⊥
4      valid ←∅
5    else
6      valid ←σ.parent.valid
7    init(σ,valid)
8    σ.V ←True
9    σ.instack ←True
10   stack.push(σ)
11   # start dfs
12   σ.children ←subgoals(σ)
13   case σ.children
14     {True} :
15       σ.flag ←True
16       ret_ltl(σ)
17     ∅ :
18       σ.flag ←False
19       ret_ltl(σ)
20     otherwise :
21       loop_ltl(σ)

1  def loop_ltl(σ) is
2    while σ.children ≠ ∅ and σ.flag != False
3      σ′ ←σ.children.pick()
4      if σ′.V
5        if not σ′.flag
6          σ.flag ←False
7        elif σ′.instack
8          σ.low ←min(σ.low, σ′.low, σ′.dfsn)
9          σ.valid ←{⟨φ1Rφ2,sp⟩ ∈ σ.valid | sp≤ σ′.dfsn}
10         if σ.valid = ∅
11           σ.flag ←False
12     else
13       # flag = dfs(σ′, σ.valid)
14       σ′.parent ←σ
15       todo.push(σ′)
16       return
17   if σ.dfsn = σ.low
18     var top ←⊥
19     while top ≠ σ
20       top ←stack.pop()
21       top.instack ←False
22       if not σ.flag
23         top.flag ←False
24   ret_ltl(σ)

1  def ret_ltl(σ) is
2    if σ.parent ≠ ⊥
3      up_ltl(σ.parent,σ)

1  def up_ltl(σ,σ′) is
2    # flag = dfs(σ′,σ.valid)
3    σ.flag ←σ′.flag
4    if σ′.low ≤ σ.dfsn
5      σ.low ←min(σ.low, σ′.low, σ′.dfsn)
6      σ.valid ←σ′.valid
7    loop_ltl(σ)
```

Figure 3.7. Sequential iterative algorithm for LTL model checking.

The main loop processes each σ in todo using the sequential checker SeqChkLTL, which is possible because the corresponding parts of the proof structure are independent (P4 properties of the previous chapter). SeqChkLTL uses subgoals (see Figure 3.9) to traverse the proof structure. For rules (R1) to (R6), the result remains local because the Petri net states do not change. However, for rule (R7), we compute separately the next states for $\mathsf{succ}_{\mathcal{L}}$ and $\mathsf{succ}_{\mathcal{R}}$: the former results in local states to be proceeded in the current step, while the latter results in states to be proceeded in the next step. If no local state is found but there exist remote states, we set subg← {True} which indicates that the local exploration succeeded (P2) and allows to proceeded

```
1  def Init_main() is
2    super_step,dfn,V,E,todo←0,0,∅,∅,∅
3    if cpu(σ_init)=mypid
4      todo←todo ∪ {σ_init}
5    flag , total←⊥,1
6
7  def Exchange(tosend,flag) is
8    dump (V, E) at super_step
9    super_step←super_step+1
10   tosend←tosend ∪ {(i,flag) | 0 ≤ i < p}
11   rcv, total←BSP_EXCHANGE(Balance(tosend))
12   flag ,rcv←filter_flag(rcv)
13   return flag, rcv, total
```

```
1  def ParChkLTL((s ⊢ Φ) as σ) is
2    def Init_main() is (...)
3    def Exchange(tosend,flag) is (...)
4    def subgoals(σ,send) is (...)
5    def Build_trace(σ) is (...)
6    Init_main()
7    while flag=⊥ ∧ total>0
8      send←∅
9      # In parallel thread,
10     # on per independent sub−part,
11     # on multi−core architectures
12     while todo ≠ ∅ ∧ flag=⊥
13       pick σ from todo
14       if σ ∉ V
15         flag←SeqChkLTL(σ,send,E,V)
16     if flag≠ ⊥
17       send←∅
18     flag ,todo,total←Exchange(send,flag)
19   case flag
20       ⊥ : print "OK"
21       σ : Build_trace(σ)
```

Figure 3.8. A BSP algorithm for LTL checking.

to the next super-step in the main loop. When all the local states have been proceeded, states are exchanged, which leads to the next slice (*i.e.* the next super-step). In order to terminate the algorithm as soon as one processor discovers a counterexample, each locally computed flag is sent to all the processors and the received values are then aggregated using function filter_flag that selects the flag with the lowest dfsn value computed on the processor with the lower number, which ensures that every processor chooses the same flag and then computes the same trace. To balance computation, we use the number of states as well as the size of the formula to be verified for each state (on which the number of subgoals directly depends).

Notice also that at each super-step, each processor dumps V and E to its local disk, recording the super-step number, in order to be able to reconstruct a trace. When a state that invalidates the formula is found, a trace from the initial state to the current σ is constructed. Figure 3.10 gives this algorithm.

The data to do so is distributed among processors and dumped into local files, one per super-step. We thus use exactly as many steps to rebuild the trace as we used to discover the erroneous state. The algorithm is presented in Figure 3.10: a trace π whose "oldest" state is σ is reconstructed following the proof graph backward. The processor that owns σ invokes Local_trace to find a path from a state σ', that was in todo at the beginning of the super-state, to σ. Then it sends σ' to its owner to let the reconstruction continue. To simplify things, we print parts of the reconstructed trace as they are locally computed. Among the predecessors of a state, we always choose those that are not yet in the trace π (set_of_trace(π) returns the set of states in π) and selects one with the minimal dfsn value (using function min_dfsn), which allows to select shorter traces.

3.4 CTL* checking

As for LTL, we first present a general algorithm and them specialised parallel algorithms for security protocols. The first one called "naive" is a first attempt to extend the parallel algorithm for LTL checking to CTL* formula whereas the second optimises (balances) the communications and reduces the number of super-steps.

```
1
2  def subgoals(σ,send) is
3    case σ
4      s ⊢ A(Φ, p) :
5          subg←if s ⊨ p then {True}
6                  else {s ⊢ A(Φ)} (R1, R2)
7      s ⊢ A(Φ, φ1 ∨ φ2) :
8        subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
9      s ⊢ A(Φ, φ1 ∧ φ2)
10       subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
11     s ⊢ A(Φ, φ1U φ2) : subg←{s ⊢ A(Φ, φ1, φ2),
12       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
13     s ⊢ A(Φ, φ1Rφ2) : subg←{s ⊢ A(Φ, φ2),
14       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
15     s ⊢ A(Xφ1, ..., Xφn) :
16       subg←{s' ⊢ A(φ1, ...φn) | s' ∈ succL(s)}
17       tosend←{s' ⊢ A(φ1, ...φn) | s' ∈ succR(s)}
18       E←E ∪ {σ ↦R σ' | σ' ∈ tosend }
19       if subg=∅ ∧ tosend≠∅
20          subg←{True}
21       send←send ∪ tosend (R7)
22   E←E ∪ {σ ↦L σ' | σ' ∈ subg }
23   return subg
```

Figure 3.9. A BSP algorithm for LTL checking.

```
1  def Build_trace(σ) is
2    def Local_trace(σ,π) is (...)
3    def Exchange_trace(my_round,tosend,π) is (...)
4    end←False
5    repeat
6      π←ϵ
7      my_round←(cpu(σ)=mypid)
8      end←(σ=σ0)
9      send←∅
10     if my_round
11       dump (V,E) at super_step
12       super_step←super_step−1
13       undump (V,E) at super_step
14       σ,π←Local_trace(σ,π)
15       π←Reduce_trace(π)
16       F←F ∪ set_of_trace(π)
17       print π
18     σ←Exchange_trace(my_round,σ)
19   until ¬end
20
21 def Exchange_trace(my_round,tosend,π) is
22   if my_round
23     tosend←tosend ∪ {(i,σ) | 0 ≤ i < p}
24   {σ},_←BSP_EXCHANGE(tosend)
25   return σ
26
27 def Local_trace(σ,π) is
28   if σ = σ0
29     return (σ,π)
30   tmp←prec(σ) \ set_of_trace(π)
31   if tmp=∅
32     σ′←min_dfsn(prec(σ))
33   else
34     σ′←min_dfsn(tmp)
35   π←π.σ′
36   if σ′ ↦R σ
37     return(σ′,π)
38   return (σ′,π)
```

Figure 3.10. BSP algorithm for building the trace after an error.

3.4.1 Sequential algorithms for CTL*

(a) Recursive algorithm [28]

The global model-checking algorithm for CTL* (named modchkCTL∗ and presented in Figure 3.11) processes a formulae P by recursively calls modchkLTL appropriately when it encounters assertions of the form $s \vdash A\Phi$ or $s \vdash E\Phi$ or by decomposing the formulae P into subformulas whose it applies to itself. The modchkCTL∗ procedure thus matches the pattern of the formulae and acts accordingly. The key idea is to use the equivalence rule of an exits-formulae with the negation of the corresponding forall-formulae to check these latter. Indeed, we already have a recursive algorithm to check LTL formula, and one can see a forall-formulae like a LTL formula by masking all elements beginning by exists or forall (let's recall that by the hypothesis, the negation precede only the atoms). Thus, when we encounter elements beginning by exists or forall, we call modchkCTL∗ to proceed in the following manner:

- if the formulae is a forall-formulae, we recursively call modchkLTL to check the validity of the subformula;
- otherwise, we use modchkLTL to check the negation of the exists-formula, the final result being the negation of the answer, in accordance to the equivalence rule between an exists-formula and its negation (the Lemma 1 ensuring this fact).

Notice that in the case of CTL* model checking, the cases of pattern of an Or-formulae and a And-formulae will be matched only at the beginning of the algorithm: indeed, otherwise these cases will be covered by the rules of the proof graph construction in subgoals (rules R3 and R4).

Note also a slight but important modification to procedure subgoals: when it encounters an assertion of the form $s \vdash A(p, \Phi)$ (notably where p is $A\varphi$ or $E\varphi$), it recursively invokes modchkCTL∗($s \vdash p$) to determine if $s \vDash p$ and then decides if rule R1 or rule R2 (of Figure 3.5) needs to be applied. In other words, one extends the atomic test in subgoals by using modchkCTL∗ procedure in the case of these subformula. We have thus a double-recursively of modchkCTL∗ and modchkLTL.

Also note, that each call to modchkLTL creates a new empty stack and a new dfn (depth-first number) since a new LTL checking is run: by abuse of language, we will named them "LTL sessions". These sessions can shared assertions which thus shared their validity (is in F or not). Take for example the following formulae: $A(pU(E(rUp)))$; there will be two LTL sessions, one for the global formulae and one for the subformula $E(rUp)))$. It is clear that the atomic proposition p would be thus test twice on the states of the Kripke structure. It can also happen for the following formulae: $A(pUq) \vee E(pUq)$. And in this second case the two sessions would also share only atomic propositions.

Thus, more subtly, LTL sessions do *not* shared their depth-first numbers (low and dfsn fields), their valid fields and thus their membership to stacks of LTL sessions. This is due because of assertions are of the form $s \vdash (\varphi_1 \vee \varphi_n)$ and also by means of the rules of Figure 3.5: these rules force that call to modchkCTL∗ within a LTL session (for checking a subforma that is not LTL and thus to have another LTL session) is perform only on a subpart of the original assertion and which is strictly smaller hence ensuring no intersection of the proof-structures (graph) of the LTL sessions ensuring that SCC are disjoints.

(b) Iterative algorithm

As previously, we now give an iterative version of the above recursive algorithm. This allow us to have only one main loop and no side-effects within recursive calls. We thus extend the LTL iterative algorithm of Section 3.3. Considering only one main loop for parallel computations has also the advantage to easily stop the computation whereas results of other processors are expected. Figure 3.12 gives this main loop.

```
1  def modchkLTL(σ) is
2    var dfn ←0
3    var stack ←ϵ
4    def init (σ,valid) is  (...)
5    def dfs(σ, valid) is  (...)
6    def subgoals(σ) is (...)
7    return dfs(σ,∅)

1  def init (σ,valid) is
2    dfn ←dfn+1
3    σ.dfsn ←σ.low ←dfn
4    σ.valid ←{⟨φ1Rφ2,sp⟩ | φ2 ∉ σ
5      ∧(φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)
6      ∧ sp=(sp′ if ⟨φ1Rφ2,sp′⟩ ∈ valid, dfn otherwise}

1  def subgoals(σ) is
2    case σ
3      s ⊢ A(Φ, p) :
4        if modchkCTL∗(s ⊨ p) then subg←{True}
5        elif Φ = ∅ then subg←∅
6        else subg← A(Φ)
7      s ⊢ A(Φ, φ1 ∨ φ2) :
8        subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
9      s ⊢ A(Φ, φ1 ∧ φ2) :
10       subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
11     s ⊢ A(Φ, φ1Uφ2) :
12       subg←{s ⊢ A(Φ, φ1, φ2),
13       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
14     s ⊢ A(Φ, φ1Rφ2) :
15       subg←{s ⊢ A(Φ, φ2),
16       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
17     s ⊢ A(Xφ1, ..., Xφn) :
18       subg←{s′ ⊢ A(φ1, ...φn) | s′ ∈ succ(s)} (R7)
19   return subg

1  def dfs(σ, valid) is
2    σ.flag ←True
3    init (σ, valid)
4    σ.V ←True
5    subg ←subgoals(σ)
6    case subg
7      {True} : σ.flag ←True
8      ∅ : σ.flag ←False
9    otherwise :
10     for σ′ ∈ subg while σ.flag
11       elif σ′.V
12         if not σ′.flag
13           σ.flag← False
14         else
15           if σ′.stack
16             σ.low ←min(σ.low,σ′.low)
17             σ.valid ←{⟨φ1Rφ2,sp⟩ ∈ σ.valid | sp≤ σ′.dfsn}
18             if σ.valid = ∅
19               σ.flag ←False
20       else
21         σ.flag ←dfs(σ′,σ.valid)
22         if σ′.low≤ σ.dfsn
23           σ.low ←min(σ.low,σ′.low)
24           σ.valid ←σ′.valid
25   if σ.dfsn = σ.low
26     var top ←⊥
27     while top ≠ σ
28       top ←stack.pop()
29       if not σ.flag
30         top.flag ←False
31   return σ.flag

1  def modchkCTL∗(σ) is
2    def modchkLTL(σ) is (...)
3    if not σ.V
4      σ.V ←True
5      case σ
6        s ⊢ p where p ∈ {a, ¬a}, a ∈ A :
7          σ.flag ←s ⊨ p
8        s ⊢ p1 ∧ p2 :
9          σ.flag ←modchkCTL∗(s ⊢ p1) ∧ modchkCTL∗(s ⊢ p2)
10       s ⊢ p1 ∨ p2:
11         σ.flag ←modchkCTL∗(s ⊢ p1) ∨ modchkCTL∗(s ⊢ p2)
12       s ⊢ Aφ :
13         σ.flag ←modchkLTL(σ)
14       s ⊢ Eφ :
15         σ.flag ←not modchkLTL(s ⊢ neg(Eφ))
16   return σ.flag
```

Figure 3.11. Sequential Recursive Algorithm for CTL* model checking.

As before, during the initialisation phase, we put the initial assertion σ_0 in the stack todo. todo would contains the assertions awaiting to be explored and it allows us the derecursification of the previous algorithm: while this set is not empty, we run the main loop which consists of two main actions:

1. First, an assertion is pick from todo (unstack) and we continue the exploration of this assertion by the call of call_ctlstar; if this assertion is visited, then we return its flag; otherwise we explore its; several cases can appear following the form of the assertion $\sigma \equiv s \vdash \varphi$:

   - if it is an atom, then we found its flag and we perform a backtracking by ret_ctlstar;
   - if it is a conjunction $\varphi \equiv \varphi_1 \wedge \varphi_2$ (resp. a disjunction $\varphi \equiv \varphi_1 \vee \varphi_2$), then this assertion will have two children: $\sigma' \equiv s \vdash \varphi_1$ and $\sigma'' \equiv s \vdash \varphi_2$; which we put in the field .children of σ;

2. then we perform a loop_ctlstar on σ to explore its children; if the assertion begins by A,

```
1  def modchkCTL*(σ0) is
2    var dfn ←0
3    var stack ←ϵ
4    var todo ←[σ0]
5    def init(σ,valid) is (...)
6    def call_ltl(σ, valid) is (...)
7    def loop_ltl(σ) is (...)
8    def ret_ltl(σ) is (...)
9    def up_ltl(σ,child) is (...)
10   def subgoals(σ) is (...)
11   def call_ctlstar(σ) is (...)
12   def loop_ctlstar(σ) is (...)
13   def ret_ctlstar(σ) is (...)
14   def up_ctlstar(σ) is (...)
15   while todo ≠ ϵ
16     σ = todo.pop()
17     call_ctlstar(σ)
18   return σ0.flag
```

Figure 3.12. Main procedure for the Iterative Algorithm for the CTL* model checking.

then the assertion must to be checked as a LTL formulae and the function call_ltl is then called on this assertion; if the assertion begins by E, this assertion has, in accordance with the algorithm, the child $\sigma' \equiv s \vdash \mathsf{neg}(E\varphi)$; a loop_ctlstar is thus called on σ' to explore this child.

To do this, we use in addition the routines call_ctlstar, loop_ctlstar, ret_ctlstar and up_ctlstar. Figure 3.13 gives the code of these routines which work as follow.

The function up_ctl as well as up_ltl propagates the answers of the checking of the formulae. This backtrack of the answers must take into account the different cases which is perform by the routine up_ctl*:

- in the case of a conjunction, $\sigma \equiv s \vdash \varphi_1 \wedge \varphi_2$, if the flag of child is True, then if the field .children is empty, each "child" of σ has carried forward an answer which is necessarily True; indeed, either the exploration would have been stopped and thus, if .children is empty, the answer of σ is True. σ has got its answer which we propagate to its parent using a call of ret_ctlstar; if the field .children of this parent is not empty, we cannot conclude since we need the answer of the other "child" wich remains to explore using a call to loop_ctlstar; if the flag of the named child is false, then the answer is false;
- the disjunction is similar;
- for the cases where σ being a forall (resp. exists), the answer of σ is the same as for its child (resp. the negation of its child);
- if $\sigma \equiv A(p \vee A(\Phi))$ then its subgoal is reduced to the singleton $\sigma' \equiv s \vdash p \vee A(\Phi)$; σ' will thus be decomposed by a call to call_ctlstar; note that p is either an atom either a formulae beginning by **forall** or **exists**; if $s \models p$ is true then σ and σ' are true; otherwise the validity of σ is reduced to the question $s \models A(\Phi)$, and this for both σ and σ'.

Note that the behaviour of this algorithm prohibits that an assertion beginning by **forall** calls an assertion beginning by **forall**. We have to take into account this fact by modifying subgoals appropriately in Figure 3.14.

The function loop_ctlstar explores the children of an assertion σ. If σ has not any children, then we perform a backtracking of the answer using a call to ret_ctlstar; otherwise, we pull one of its children (named "child") and we put it into the stack todo to be explored later (we recall that todo contains the assertions awaiting for exploration); we also tag the field .parentCTL* of this child to child.parentCTL* $= \sigma$, which allow us to recover the parent of any assertion. Note

```
1  def call_ctlstar(σ) is
2    if σ.V
3      return σ.flag
4    else
5      σ.V ←True
6      case σ
7        s ⊢ p where p ∈ {a, ¬a}, a ∈ A :
8          σ.flag ←s ⊨ p
9          ret_ctlstar(σ)
10       s ⊢ φ1 ∧ φ2 :
11         σ.wait ←σ.children ←{s ⊢ φ1, s ⊢ φ2}
12         loop_ctlstar(σ)
13       s ⊢ φ1 ∨ φ2 :
14         σ.wait ←σ.children ←{s ⊢ φ1, s ⊢ φ2}
15         loop_ctlstar(σ)
16       s ⊢ A(φ) :
17         call_ltl (σ)
18       s ⊢ E(φ) :
19         σ.children ←{s ⊢neg(Eφ)}
20         loop_ctlstar(σ)

1  def loop_ctlstar(σ) is
2    if σ.children ≠ ∅
3      child ←σ.children.pop()
4      child.parentCTL∗ ←σ
5      todo.push(child)
6    else
7      ret_ctlstar(σ)

1  def ret_ctlstar(σ) is
2    if σ.parentCTL∗ ≠ ⊥
3      up_ctl∗(σ.parentCTL∗, σ)
4    elif σ.parentLTL ≠ ⊥
5      ret_ltl(σ)

1  def up_ctlstar(σ,child) is
2    case σ
3      s ⊢ φ1 ∧ φ2 :
4        if child.flag = True
5          if σ.children = ∅
6            σ.flag = True
7            ret_ctlstar(σ)
8          else
9            loop_ctlstar(σ)
10       else # child.flag = False
11         σ.children = ∅
12         σ.flag ←False
13         ret_ctlstar(σ)
14     s ⊢ φ1 ∨ φ2 :
15       if child.flag = True
16         σ.children = ∅
17         σ.flag ←True
18         ret_ctlstar(σ)
19       else # σ.flag ←False
20         if σ.children = ∅
21           σ.flag = False
22           ret_ctlstar(σ)
23         else
24           loop_ctlstar(σ)
25     s ⊢ Aφ :
26       σ.flag ←flag
27       ret_ctlstar(σ)
28     s ⊢ Eφ :
29       σ.flag = not child.flag
30       ret_ctlstar(σ)
```

Figure 3.13. CTL* decomposition part for the Iterative Algorithm for the CTL* model checking.

that the field .parentCTL∗ is either $\perp$ if this assertion is the initial assertion of the algorithm modchkCTL∗ either $\neq \perp$. In this last case, it is run from the decomposition of a formulae disjunctive, conjunctive or beginning by E or A.

The function ret_ctlstar propagates the possible answers: each assertion σ will be explored either in loop_ctlstar either in call_ltl. As appropriate, the field .parentCTL∗ (resp. .parentLTL) will be filled, the other worthing to $\perp$. We have now three cases:

1. if the field .parentCTL∗ is not $\perp$, then we propagate the answer of σ to its father *via* a call of up_ctlstar; otherwise we perform a ret_ltl;

2. if the field .parentLTL is not $\perp$ and then we propagate the answer of σ to its father *via* call of up_ltl;

3. otherwise we unstack todo and we run ret_ltl.

These routines as well as those which deals with LTL checking are defined in Figure 3.15. They work as follow.

The unstacking of the stack allow us to take into account the case where the proof-structure is not reduced to a single assertion. Indeed, the propagation's chain of the answer follows a scheme base on the following sequence: ret_ltl →up_ltl →loop_ltl. But, in the case of a proof-structure reduced to a point, a such propagation in this algorithm will not be executed since the call of up_ltl following by ret_ltl allows the connexion between a child assertion and its father's call. To

```
1  def subgoals(σ) is
2    case σ
3      s ⊢ A(Φ, p) ,p ∈ A or p = ∗φ and ∗ ∈{A, E}:
4        subg← {s ⊢ p ∨ A(Φ)}
5      s ⊢ A(Φ, φ1 ∨ φ2) :
6        subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
7      s ⊢ A(Φ, φ1 ∧ φ2) :
8        subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
9      s ⊢ A(Φ, φ1Uφ2) :
10       subg←{s ⊢ A(Φ, φ1, φ2),
11       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
12     s ⊢ A(Φ, φ1Rφ2) :
13       subg←{s ⊢ A(Φ, φ2),
14       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
15     s ⊢ A(Xφ1, ..., Xφn) :
16       subg←{s′ ⊢ A(φ1, ...φn) | s′ ∈ succ(s)} (R7)
17   return subg
```

Figure 3.14. Subgoals procedure for the Iterative Algorithm for the CTL* model checking.

do so, the call of loop_ltl (by up_ltl) unstack the LTL's exploration's stack — as before, named stack. On the case of a proof-graph reduced to a single node, the single explored assertion will not be unstacked and the sequence ret_ltl →up_ltl →loop_ltl is not performed because this assertion have not a field .parentLTL: this is the initial assertion of a LTL session, and therefore its field .parentLTL must be ⊥ — it is not executed in the run of a LTL exploration.

Otherwise, the propagation of the answer until the initial assertion of the ongoing LTL session has necessarily be done by a sequence constituted of the repetition of the following scheme: ret_ltl →up_ltl →loop_ltl. The last assertion following this sequence is precisely the initial assertion of the LTL session. For example, considered the sequence ret_ltl(σ') →up_ltl(σ, σ') →loop_ltl(σ) for any σ and where σ' is the intial assertion. Assertion σ has necessarily a field .dfsn identical to its field .low and it has the smallest dfsn and the stack is empty.

We give here a simple trace of execution of this algorithm. Considering the Kripke structure whose the only nodes are s and s' interconnected by the arcs $s \rightarrow_L s'$ and $s' \rightarrow_L s'$ and where $s' \not\vDash p$ and $s' \vDash q$. We want to check that $\sigma \equiv s \vDash A(Xp \vee A(Xq))$. Let's note $\sigma_0 \equiv s \vdash A(Xp \vee A(Xq))$, $\sigma_1 \equiv s \vdash A(Xp)$, $\sigma_2 \equiv s \vdash A(A(Xq))$, $\sigma'_1 \equiv s' \vdash A(p)$, $\sigma'_2 \equiv s \vdash A(Xq)$ and $\sigma''_2 \equiv s' \vdash A(q)$. We have the following trace:

```
1  par_modchkCTL∗(σ0)
2
3    −− SUPER STEP 1 −−
4    todo = [σ0]
5    call_ctlstar (σ0)
6      σ0.children = {σ1, σ2}
7      loop_ctlstar(σ0)
8        σ0.children = {σ2}
9        todo = [σ1]
10   call_ctlstar (σ1)
11     call_ltl (σ1)
12       stack = [σ1]
13       σ1.children ←{σ′1}
14       loop_ltl(σ1)
15         todo = [σ′1] and σ1.children ←∅
16         ret_ctlstar (σ1)
17   call_ctlstar (σ′1)
18     call_ltl (σ′1)
19       stack = [σ1, σ′1]
20       σ′1.flag = False
21       ret_ltl (σ′1)
22         up_ltl(σ1, σ′1)
23           σ1.flag = False
24           loop_ltl(σ1)
25             stack = ∅
26             ret_ltl (σ1)
27               ret_ctlstar(σ1)
28                 up_ctlstar(σ0, σ1)
29                   σ0.children = {σ2}
30                   loop_ctlstar(σ0)
31                     σ0.children = {}
32                     todo = [σ2]
33   call_ctlstar (σ2)
34     call_ltl (σ2)
35       stack = [σ2]
36       σ2.children = {σ′2}
37       loop_ltl(σ2)
38         σ2.children = ∅
39         todo = [σ′2]
40   call_ctlstar (σ′2)
41       call_ltl (σ′2)
42       stack = [σ2, σ′2]
```

```
1  def init(σ,valid) is
2    dfn ←dfn+1
3    σ.dfsn ←σ.low ←dfn
4    σ.valid ←{⟨φ1Rφ2,sp⟩ | φ2 ∉ σ
5      ∧(φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)
6      ∧ sp=(sp' if ⟨φ1Rφ2,sp'⟩ ∈ valid, dfn otherwise}

1  def call_ltl(σ) is
2    # init
3    if σ.parentLTL = ⊥
4      valid ←∅
5    else
6      valid ←σ.parentLTL.valid
7    init(σ,valid)
8    σ.V ←True
9    σ.instack ←True
10   stack.push(σ)
11   # start dfs
12   σ.children ←subgoals(σ)
13   case σ.children
14     {True} :
15       σ.flag ←True
16       ret_ltl(σ)
17     {⊥} :
18       σ.flag ←⊥
19       ret_ltl(σ)
20     ∅ :
21       σ.flag ←False
22       ret_ltl(σ)
23     otherwise :
24       loop_ltl(σ)

1  def loop_ltl(σ) is
2    while σ.children ≠ ∅ and σ.flag ≠ False
3      σ' ←σ.children.pick()
4      if σ'.V
5        if σ'.flag = False
6          σ.flag ←False
7        elif σ'.instack
8          σ.low ←min(σ.low, σ'.low, σ'.dfsn)
9          σ.valid ←{⟨φ1Rφ2,sp⟩ ∈ σ.valid | sp≤ σ'.dfsn}
10         if σ.valid = ∅
11           σ.flag ←False
12     else
13       # flag = dfs(σ', σ.valid)
14       σ'.parentLTL ←σ
15       todo.push(σ')
16       return
17   if σ.dfsn = σ.low
18     var top ←⊥
19     while top ≠ σ
20       top ←stack.pop()
21       top.instack ←False
22       if not σ.flag
23         top.flag ←False
24   ret_ltl(σ)

1  def ret_ltl(σ) is
2    if σ.parentLTL ≠ ⊥
3      up_ltl(σ.parentLTL, σ)
4    else
5      stack.pop() (if stack ↯)
6      ret_ctlstar(σ)

1  def up_ltl(σ,σ') is
2    # flag = dfs(σ',σ.valid)
3    σ.flag ←σ'.flag
4    if σ'.low ≤ σ.dfsn
5      σ.low ←min(σ.low, σ'.low, σ'.dfsn)
6      σ.valid ←σ'.valid
7    loop_ltl(σ)
```

Figure 3.15. LTL part for the Iterative Algorithm for the CTL* model-checking.

```
43         σ'2.children = {σ''2}
44         loop_ltl(σ''2)
45           σ''2.children = ∅
46           todo = [σ''2]
47       call_ctlstar(σ''2)
48         call_ltl(σ'2)
49         stack = [σ2, σ'2, σ''2]
50         σ'2.flag = True
51         ret_ltl(σ''2)
52           up_ltl(σ'2, σ''2)
53             σ'2.flag = True
54             loop_ltl(σ'2)

55               stack = [σ2]
56               ret_ltl(σ'2)
57                 up_ltl(σ2, σ'2)
58                   σ2.flag = True
59                   loop_ltl(σ2)
60                     stack = [σ2]
61                     ret_ltl(σ2)
62                       ret_ctlstar(σ2)
63                         up_ctlstar(σ0, σ2)
64                           σ0.flag = True
65                           ret_ctlstar(σ0)
```

3.4.2 Naive parallel algorithm for CTL*

(a) Main loop

Here we give a first and naive attempt of parallalelisation of the iterative algorithm for CTL* model-checking. We call this algorithm "naive" because it would implies a large number of super-steps mainly equivalent to the number of states in the Kripke structure — depending of the CTL* formulae.

The algorithm works as follow. As before, a main loop (which computes until an empty stack

```
1  var slice ←0
2  var V ←∅
3  var F ←∅
4
5  def modchkCTL∗(σ0) is
6    if not σ.V
7      σ.V ←True
8      case σ
9    s ⊢ p where p ∈ {a, ¬a}, a ∈ A :
10     σ.flag ←s ⊨ p
11   s ⊢ p1 ∧ p2 :
12     σ.flag ←modchkCTL∗(s ⊢ p1)
13                ∧ modchkCTL∗(s ⊢ p2)
14   s ⊢ p1 ∨ p2:
15     σ.flag ←modchkCTL∗(s ⊢ p1)
16                ∨ modchkCTL∗(s ⊢ p2)
17   s ⊢ Aφ :
18     σ.flag ←par_modchkCTL∗(σ)
19   s ⊢ Eφ :
20     σ.flag ←not par_modchkCTL∗(s ⊢ neg(Eφ))
21   return σ.flag
```

```
1  def par_modchkCTL∗(σ0) is
2    var out_stack ←ϵ
3    var answer_ltl, flag_list , mck
4    var σ ←σ0
5    repeat
6      if σ ≠ ⊥
7        mck ←new LTL(σ)
8        flag_list ,σ ←mck.explore()
9        out_stack.push(mck)
10     else
11       if flag_list ≠ ∅
12         answer_ltl ←False
13         mck ←out_stack.top()
14         mck.updateF(flag_list)
15       else
16         answer_ltl ←True
17       out_stack.pop()
18       if out_stack ≠ ∅
19         mck ←out_stack.top()
20         flag_list ,σ ←mck.recovery(answer_ltl)
21   until out_stack = ∅
22   return answer_ltl
```

Figure 3.16. Main procedures for the Naive Algorithm for parallel CTL* model-checking.

todo) is used but in a SPMD fashion: each processor performs this main loop. Figure 3.16 gives the code of this main loop. The algorithm first uses the procedure modchkCTL∗ to decompose the initial formulae and run par_modchkCTL∗ which contain the main loop. Then, during the decomposition of the CTL* formulae or during the subgoals of a the computations of the SCC of the proof-structures (see Figure 3.21), when a subformulae beginning by A or E is found, the computation is halting and a new "session" is run for this assertion using the par_modchkCTL∗ routine — which here only supports CTL* formula beginning by forall operator A. The ongoing "session" is now halting and is waiting for the answer of the new session based on the appropriate assertion. par_modchkCTL∗ is a kind of sequential algorithm but implicitly parallel because it runs parallel sessions.

par_modchkCTL∗ routine uses a stack named out_stack which not contains assertions but "LTL objects" parametrised by an assertion beginning by forall A. These LTL objects are what we call "sessions": mainly a LTL computation as in the Section 3.3 — we give it in Figure 3.19. These objects are defined (as a class) in Figures 3.17 and 3.18.

We also mask some subtleties and strategic choices performed during the communication of the session: indeed, assume that several assertions to be tested are found on several machines, then, only one of these assertions will be returned. This naive approach is based on an "depth" exploration of the sessions: in each slice, we explore or backtrack a single session. A choice must to be done on the returned assertion, the other remaining in the memory's environment of the session, encapsulated in the LTL object. Also the balance of the assertions over the processors is done dynamically at each slice of each session: this ensures that two assertions with the same hash are contained by the same processor (for correctness of the algorithm). This also implies that the sweep-line technical used in the previous chapter could not holds or more precisely each slice does not correspond to a super-step and thus during backtracking of the answer, the save on disks assertions must be entered in the main memory.

(b) Methods of LTL objects (*i.e.* sessions)

The method .explore of a "LTL object" (a session) generates in a parallel way the proof-structure whose initial assertion is the one given as parameter and stop when:

```
1  class LTL is
2    var self.stack
3    var self.send
4    var self.dfn
5    var self. slice_init
6    var self.σ_init
7    var self.todo ←∅
8
9    # saving environment
10   var self.sigma_torecover
11   var self.valid_torecover
12   var self.σ_totest_tosauv
13
14   def init_class(σ) is
15     σ_init ←σ
16     slice_init ←slice
17     dfn ←0
18     stack ←ϵ
19     send ←∅
20     sigma_torecover ←⊥
21     valid_torecover ←∅
22     σ_totest_tosauv ←⊥
23
24   def next_slice() is
25     dump (V,F,slice)
26     slice ←slice+1
27     undump (V,F,slice)
28
29   def previous_slice() is
30     dump (V,F,slice)
31     slice ←slice−1
32     undump (V,F,slice)
```

Figure 3.17. LTL class for the Naive Algorithm for parallel computing of the CTL* model checking — part 1.

```
1    def sub_explore() is
2      var σ_totest ←⊥
3      var flag ←⊥
4      var flag_list ←∅
5      while self.todo and not flag=⊥
6                        and not σ_totest = ⊥
7        var σ ←self.todo.pop()
8        if σ in F
9          flag ←{σ}
10       elif σ ∉ V
11         flag,σ_totest ←dfs(σ,∅)
12     return flag,σ_totest
13
14   def explore() is
15     var total ←1
16     var σ_totest ←⊥
17     var flag ←⊥
18     var flag_list ←∅
19
20     if cpu(σ_init) = my_pid:
21       self.todo ←self.todo ∪ {σ_init}
22
23     while not flag_list and total>0 and σ_totest ≠ ⊥ is
24       self.send ←∅
25       flag,σ_totest ←sub_explore()
26       next_slice()
27       flag_list, total, σ_totest ←
28                        BSP_EXCHANGE(flag, σ_totest)
29
30     previous_slice()
31     if σ_totest = s ⊢ ∗φ ≠ ∅, ∗ ∈ {A,E}
32       if ∗ ≡ E then σ_totest ←s ⊢ neg(∗φ)
33     return flag_list, σ_totest
34
35   def recovery(answer_ltl) is
36     if σ_totest_tosauv = p ⊢ Eφ and answer_ltl = True
37       F ←F ∪ {σ_totest_tosauv}
38       V ←V ∪ {σ_totest_tosauv}
39
40     var σ_totest ←⊥
41     flag ←⊥
42     flag_list ←∅
43
44     #le processeur qui possedait le sigma
45     if cpu(σ) = my_pid
46       flag,σ_totest ←dfs(σ,sauv_valid)
47
48       while todo and not flag and σ_totest ≠ ⊥
49         σ = todo.pop()
50
51         if σ in F
52           flag ←{σ}
53         elif σ ∉ V
54           flag,σ_totest ←dfs(σ,∅)
55
56     next_slice()
57     flag_list, total, σ_totest ←BSP_EXCHANGE(flag,σ_totest)
58
59     # back to normality
60     while not flag_list and total>0 and σ_totest ≠ ⊥
61       self.send ←∅
62       flag,σ_totest ←sub_explore()
63       next_slice()
64       flag_list, total, σ_totest ←BSP_EXCHANGE(flag,σ_totest)
65
66     previous_slice()
67     if σ_totest = s ⊢ ∗φ ≠ ∅, ∗ ∈ {A,E}
68       if ∗ ≡ E then σ_totest ←s ⊢ neg(∗φ)
69     return flag_list, σ_totest
```

Figure 3.18. LTL class for the Naive Algorithm for parallel computing of the CTL* model checking — part 2.

- a subformulae $\varphi \in \sigma$ of an assertion $\sigma \equiv s \vdash \{\Phi, *\varphi\}$ ($* \in A$ or E) is found, then the return value is ([], $s \vdash *\varphi$); this first case corresponds to a halting of the current session;

- otherwise, if the assertion is checked truly, then the return value will be ([], $\perp$), else some assertions $\sigma_1, ..., \sigma_k$ invalid the ongoing computation, *i.e.* the initial assertion; the returning value will be thus ($[\sigma_1, ..., \sigma_k]$, $\perp$). Note that the returned assertion corresponds to its validity.

The method .recovery resumes the computation by passing as an argument the boolean value corresponding to the validity of the assertion previously returned — and awaiting to test. This boolean is an answer corresponding to the test of validity required on the assertion returned by .explore. Thus, as for the method .explore, if the assertion is not checked the method .recovery returns the assertions invalidating the ongoing computation. More precisely, the backtracking was already performed during the last computed slice, in accordance to the state-space algorithm. It remains to continue the backtracking from the assertions $\sigma_1, ..., \sigma_k$ on the previous slices until the initial slice, *i.e.* the slice of the initial assertion of the ongoing session. This recovery of the backtracking is performed by the method .updateF which, as its name indicates, updates the set F of the false assertions.

The following variables are also used during the computation of the main loop:

- out_stack manages the "depth" exploration of the sessions by storing the LTL object and is initially empty;
- answer_ltl saves the answer (True or False) when a LTL session is finished;
- flag_list contains the assertions infringing the computation and is used for the backtracking;
- mck represents the LTL object to use (exploration, computation's recovery, backtracking);
- at least, σ represents the assertion to test through a new LTL session (*via* a new LTL object, which is instantiated from of this assertion σ).

During the main loop, two cases may happen.

1. First, if the variable σ contains an assertion, *i.e.* is not $\perp$, then we create an LTL object of this assertion, we explore it and we stack it in the stack's session out_stack; otherwise the computation of the last object pushed into the stack out_stack is finished.

2. If the variable flag_list is not empty, the answer is false and one must do the backtracking *via* the method .updateF on this last pushed object (presented in Figure 3.20); otherwise, the answer of the session is true.

The computation of the last LTL object found is now completely finished in the sense where if the answer is false, the backtracking was already performed. We are therefore on the slice corresponding to the last slice of the ongoing computation of the penultimate LTL object stacked. The computation of the last LTL object being completely finished, we unstack it.

If the stack of the sessions is not empty, then we resume the computation of the last session's object stacked by the method .recovery in which one put, as argument, the answer of the session found beforehand. This one is currently, the answer of a test required by the session henceforth in progress. The answer of par_modchkCTL$*$ is the value of the variable answer_ltl when the stack is empty, *i.e.* the answer of the assertion given as parameter to par_modchkCTL$*$.

(c) Example

As example, we use a simple Kripke structure which contains only $s \vdash p$ (with an arc pointing to oneself) and the CTL* formulae $EAEp$. The following gives the running trace. The parallel feature of the algorithm is induced only by the parallel aspect of the LTL's algorithm underlying. In this way, the global shape of the algorithm is sequentially considering a depth first exploration of the sessions. Thus the execution of the algorithm in our example masks the parallel feature which is contained in the LTL parallel algorithm. Figure 3.22 gives an overview of the sequence of LTL sessions induces by this example:

```
1  def init(σ,valid) is
2    dfn ←dfn+1
3    σ.dfsn ←σ.low ←dfn
4    σ.valid ←{⟨φ1Rφ2,sp⟩ | φ2 ∉ σ
5      ∧(φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)
6      ∧ sp=(sp′ if ⟨φ1Rφ2,sp′⟩ ∈ valid, dfn otherwise}
7
8   def dfs (σ,valid) is
9    var flag ←⊥
10   var σ_totest ←⊥
11   var subg
12   init(σ,valid)
13   stack ←stack ∪ {σ}
14   V ←V ∪ {σ}
15   subg,σ_totest ←σ.subgoals(self.send)
16
17   if σ_totest # saving environment
18     self.sigma_torecover ←σ
19     self.valid_torecover ←σ.valid
20     self.σ_totest_tosauv ←σ_totest
21     return subg,σ_totest
22
23   if subg = {True}
24     pass
25   elif subg = ∅
26     flag ←{σ}
27   else
28     for σ′ ∈ subg
29       if flag
30         break
31       if σ′ ∈ V
32         if σ′ ∈ F
33           flag ←{σ}
34         elif σ′ ∈ stack
35           σ.low ←min(σ.low, σ′.low)
36           σ.valid ←{⟨φ1Rφ2,sp⟩ ∈ σ.valid | sp≤ σ′.dfsn}
37           if σ.valid = ∅
38             flag ←σ
39       else
40         valid ←σ.valid
41         flag,σ_totest ←dfs(σ′,σ.valid)
42         if σ_totest
43           return subg,σ_totest
44         if σ′.low ≤ σ.dfsn
45           σ.low ←min(σ.low,σ′.low)
46           σ.valid ←σ′.valid
47   if σ.dfsn = σ.low
48     while True
49       σ′ ←stack.pop()
50       if flag
51         F ←F ∪ σ′
52       if σ = σ′ :
53         break
54   return flag,σ_totest
```

Figure 3.19. LTL class for the Naive Algorithm for parallel computing of the CTL* model checking — part 3.

```
1   def updateF(flag_list) is
2     previous_slice()
3     var end ←False
4     var send_flag
5
6     while slice ≥ slice_init
7       send_flag ←∅
8
9       while flag_list
10        σ ←flag_list.pop()
11        π ←∅
12        σ′,π ←local_trace(σ,π)
13        F ←F ∪ π
14        if σ′ ≠ 0
15          send_flag.add(sgm_nxt)
16
17      previous_slice()
18      flag_list ←exchange_trace(send_flag)
19
20    self.next_slice()
21
22  def local_trace(σ,π) is
23    if σ = σ_init
24      return σ,π
25    prec ←∅
26    prec.update(σ.prec_L)
27    prec.update(σ.prec_R)
28    prec ←prec − F
29    if not prec
30      return 0, π
31    σ′ ←prec.pop()
32    π ←π ∪ σ′
33    if σ′ in σ.prec_R
34      return σ′,π
35    return local_trace(σ′,π)
```

Figure 3.20. (Backtracking part) LTL class for the Naive Algorithm for parallel computing of the CTL* model checking — part 4.

```
1  par_modchkCTL*(s ⊢ EAEp)
2    out_stack = ϵ
3    σ = s ⊢ EAEp
4    −− LOOP ITERATION #1 −−
5    mck1 = LTL(σ)
6    flag_list = ϵ, σ = s ⊢ AEA¬p
7    out_stack = [mck1]
8    −− LOOP ITERATION #2 −−
9    mck2 = LTL(σ)
10   flag_list = ϵ, σ = s ⊢ EA¬p
11   out_stack = [mck1, mck2]
12   −− LOOP ITERATION #3 −−
13   mck3 = LTL(σ)
14   flag_list = ϵ, σ = s ⊢ AEp
15   out_stack = [mck1, mck2, mck3]
16   −− LOOP ITERATION #4 −−
```

```
1  def subgoals(σ) is
2    var σ_totest ←⊥
3    var subg ←∅
4    case σ
5      s ⊢ A(Φ, p) :
6        if p ≡ ∗φ, ∗ ∈ {A, E} and s ⊢ ∗φ ∉ is known
7          subg ←∅
8          σ_totest ←s ⊢ ∗φ
9        elif s ⊨ p or s ⊨ p ∉ F then subg ←{True}
10       elif Φ = ∅ then subg ←∅
11       else subg ←A(Φ)
12     s ⊢ A(Φ, φ1 ∨ φ2) :
13       subg ←{s ⊢ A(Φ, φ1, φ2)} (R3)
14     s ⊢ A(Φ, φ1 ∧ φ2) :
15       subg ←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
16     s ⊢ A(Φ, φ1Uφ2) :
17       subg ←{s ⊢ A(Φ, φ1, φ2),
18       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
19     s ⊢ A(Φ, φ1Rφ2) :
20       subg ←{s ⊢ A(Φ, φ2),
21       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
22     s ⊢ A(Xφ1, ..., Xφn) :
23       subg ←{s' ⊢ A(φ1, ...φn) | s' ∈ succ(s)} (R7)
24   return subg,σ_totest
```

Figure 3.21. Subgoal procedure for the Naive Algorithm for parallel computing of the CTL* model checking.

```
17  mck4 = LTL(σ)
18   flag_list = ϵ, σ = s ⊢ Ep
19  out_stack = [mck1, mck2, mck3, mck4]
20  −− LOOP ITERATION #5 −−
21  mck5 = LTL(σ)
22   flag_list = ϵ, σ = s ⊢ A¬p
23  out_stack = [mck1, mck2, mck3, mck4, mck5]
24  −− LOOP ITERATION #6 −−
25  mck6 = LTL(σ)
26   flag_list = [p ⊢ A¬p], σ = ⊥
27  out_stack = [mck1, mck2, mck3, mck4, mck5, mck6]
28  −− LOOP ITERATION #7 −−
29  answer_ltl = False
30  mck6.updateF([s ⊢ A¬p])
31  out_stack = [mck1, mck2, mck3, mck4, mck5]
32  mck5.recovery(False)
33   flag_list = ϵ, σ = ⊥
34  −− LOOP ITERATION #8 −−
35  answer_ltl = True
36  out_stack = [mck1, mck2, mck3, mck4]
37  mck4.recovery(True)
38   flag_list = ϵ, σ = ⊥
39  −− LOOP ITERATION #9 −−
40  answer_ltl = True
41  out_stack = [mck1, mck2, mck3]
42  mck4.recovery(True)
43   flag_list = [s ⊢ EA¬p], σ = ⊥
44  −− LOOP ITERATION #10 −−
45  answer_ltl = False
46  mck3.updateF([s ⊢ A¬p])
47  out_stack = [mck1, mck2]
48  mck2.recovery(False)
49   flag_list = [s ⊢ AEA¬p], σ = ⊥
50  −− LOOP ITERATION #11 −−
51  answer_ltl = False
52  mck2.updateF([s ⊢ AEA¬p])
53  out_stack = [mck1]
54  mck1.recovery(False)
55   flag_list = ϵ, σ = ⊥
56  −− LOOP ITERATION #12 −−
57  answer_ltl = True
58  out_stack = ϵ
```

Figure 3.23 illustrates the progress of our naive parallel algorithm for CTL* checking. The development of a "new session" means that we initiate the generation (in a parallel way) of a proof graph for the checking of an assertion where the formula begin by a "forall" A. During the exploration of this proof graph, when we encounter a subgoal beginning by a "forall" A, we pause the exploration of the current session to start (in a parallel way) the new session corresponding to this subgoal. But it is possible to encounter several subgoals beginning by a "forall" on several machines. In this case we choose to start only one "new session" corresponding to a single subgoal of a particular assertion.

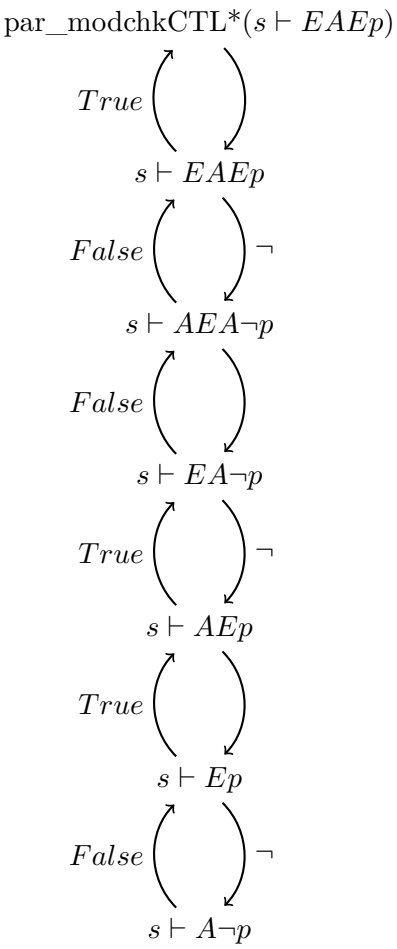


Figure 3.22. Answer's scheme, *i.e.* sequence of sessions of the running example of the main procedures for the Naive parallel algorithm for CTL* model checking.

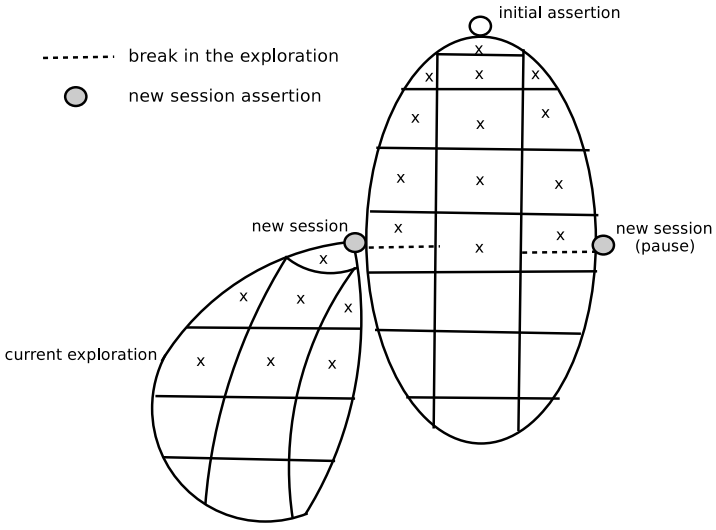


Figure 3.23. Illustration for the naive parallel algorithm for CTL* checking.

3.4.3 Parallel algorithm for CTL*

(a) Problem of the previous algorithm

The previous has several defects.

First, in the case of a formulae of the form AAp, the number of super-steps would be proportional to the number of states of the Kripke structure. This is due to the fact that the algorithm works as follow for this formulae: for each state, test if Ap is valid; thus, run each time a LTL session which would implies several super-steps to test Ap (if p is valid on all the states of the Kripke structure?).

Second, each time a LTL session traverses a subpart of the Kirpke structure, only a subpart of the assertions are generated: we do not thus have all the informations for a good balancing of the computation for the next slice or super-step; this implies a partial balancing of the assertions. To remedy to this problem, two solutions can be introduced: (1) re-balancing the assertions which can imply too many communications; (2) keep these partial balancing and distribute the new found assertions following these partial balancing and full them. For convenience, we have chosen the second solution but as expected, it does not give good scalability and performances — mainly due to the huge number of super-steps.

Third, the algorithm does take into account the "nature" of the proof-structure: we have an explicit decomposition of the logical formulae which can help to choose where a parallel computation is needed or not.

(b) Main idea

The main idea of the algorithm is based on the computation of the two followings rules of the proof-structures:

$$\frac{s \vdash A(\Phi, \varphi)}{true}\ (R1)\ \text{if } s \vDash \varphi \qquad \frac{s \vdash A(\Phi, \varphi)}{s \vdash A(\Phi)}\ (R2)\ \text{if } s \nvDash \varphi$$

In the LTL formulas, φ is an atomic proposition, which can thus be sequentially computed. But in a CTL* formulae, φ can be any formulae. In the naive algorithm, we thus run another LTL computations by recursively call modchkCTL∗. The trick (heuristic) proposed for this new algorithm is to compute both $s \vDash \varphi$ (resp. and $s \nvDash \varphi$) and $s \vdash A(\Phi)$. In this way, we will able to choice which rule (R1 or R2) can be applied. As above, the computation of $s \vDash \varphi$ would be performed by a LTL session while the computation of $s \vdash A(\Phi)$ would be performed by following the execution of the sequential Tarjan algorithm — SCC computation. In a sense, we expect the result of $s \vDash \varphi$ by computing the validity of the assertion $s \vdash A(\Phi)$.

This has three main advantages:

1. as we computed both $s \vDash \varphi$ and $s \vdash A(\Phi)$, we would aggregate the super-steps of the both computations and thus reduced the number to the max;

2. we also aggregated the computations and the communications (*en masses*) without unbalanced them; similarly, we would have all the assertions (and more) of each slice, which implies a better balance of the computation than a partial balance of the naive algorithm;

3. the computation of the validity of $s \vdash A(\Phi)$ can be used latter in different LTL sessions.

On the other side, the pre-computation of $s \vdash A(\Phi)$ may well be unnecessary, but, if we suppose a sufficient number of processors, this is not a problem for scalability: the exploration is thus in a breadth fashion allow us a highest degree of parallelization.

Figure 3.24 gives the main procedure and thus the main loop. It works as follow: the computations is performed until the answer of the initial assertion is computed — the variable **done**. Not that we add another trick: we iterate in parallel over the set of received classes (computed

```
1  def par_modchkCTL*(σ0) is
2    var dfn ←0
3    var stack ←ϵ
4    var snd_todo ←∅
5    var snd_back ←∅
6    var rcv ←{σ0} if local(σ0) else ∅
7    var back ←∅
8    var done ←False
9    var todo ←∅
10   def init (σ,valid) is (...)
11   def call_ltl (σ, valid) is (...)
12   def loop_ltl(σ) is (...)
13   def ret_ltl (σ) is (...)
14   def up_ltl(σ,child) is (...)
15   def subgoals(σ) is (...)
16   def call_ctlstar (σ) is (...)
17   def loop_ctlstar(σ) is (...)
18   def ret_ctlstar (σ) is (...)
19   def up_ctlstar(σ) is (...)
20   def ret_trace(σ) is (...)
21   def up_trace(σ,child) is (...)
22   while not done
23     for_par rcv′ in split_class(rcv) while not done
24       for σ in rcv′ while not done
25         if not σ.V
26           todo ←[σ]
27           while todo and not done
28             σ ←todo.pop()
29              call_ctlstar (σ)
30     done,back,rcv ←BSP_EXCHANGE(done,snd_back,snd_todo)
31     while back and not done
32       σ, child ←back.pop()
33       up_trace(σ, child)
34     sweep()
35   return σ0.flag
```

Figure 3.24. Main procedure of CTL* model-checking algorithm.

by split) instead of sequentially iterate over all the assertions of received classes (we recall that assertions are ranged over classes that are distributed across processors); in a hybrid parallel machine (cluster of multi-cores machines), each class of each LTL session could be assigned to a core since computations of classes and sessions are independants; note that in our current implementation, we do not used this possible trick — see future work.

The main loop can be divided into two phases. First, the actual exploration, secondly, the propagation of the backtrack of the answer (not equal to $\perp$) found especially on other machines. Note that in the first stage some backtracks of answer can also be performed but they are done during the ongoing exploration. Between these two phases, an exchange between the machines is performed. This second phase, the exchange is put into the two main stages because thus, the treatment of the backtrack of the answers will be performed before the explorations, allowing that the assertions awaiting to be explored found eventually their answers, making unnecessary their exploration. The end of the main loop is marked by a partial sweep-line on the assertions in memory by the function sweep whose the description is give latter.

Note also another obvious trick: if φ in $s \vDash \varphi$ is an atomic proposition, we immediately computed its validity.

```
1  def subgoals(σ) is
2    case σ
3      s ⊢ A(Φ, p) ,p ∈ 𝒜 or p = ∗φ and ∗ ∈{A, E}:
4        subg← {s ⊢ p ∨ A(Φ)}
5      s ⊢ A(Φ, φ1 ∨ φ2) :
6        subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
7      s ⊢ A(Φ, φ1 ∧ φ2) :
8        subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
9      s ⊢ A(Φ, φ1Uφ2) :
10       subg←{s ⊢ A(Φ, φ1, φ2),
11       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
12     s ⊢ A(Φ, φ1Rφ2) :
13       subg←{s ⊢ A(Φ, φ2),
14       s ⊢ A(Φ, φ1, X(φ1Rφ2))} (R6)
15     s ⊢ A(Xφ1, ..., Xφn) :
16       subg←{s′ ⊢ A(φ1, ...φn) | s′ ∈succ_L(s)}
17       tosend ←{s′ ⊢ A(φ1, ...φn) | s′ ∈succ_R(s)}
18       ∀σ′ ∈ tosend, σ′.pred ←σ′.pred∪{σ}
19       if subg=∅ ∧ tosend ≠∅
20          subg←{⊥}
21       snd_todo←snd_todo ∪ tosend (R7)
22   ∀σ′ ∈ subg, σ′.pred ←σ′.pred∪{σ} (if subg ≠ {True})
23   return subg
```

Figure 3.25. Subgoal procedure for the Algorithm for parallel computing of the CTL* model checking.

(c) Technical modifications

To take advantage of this new algorithm, we modify the recursive algorithm doing the sequential CTL* model checking in an iterative fashion by the derecursification of the algorithm. This has been presented in Section 3.4.1.

We also modify the function subgoals (see Figure 3.25) to take into account the management of the sends, like our parallel algorithm for the LTL model checking. Also, we add arcs between the assertions, via the field .pred for each assertion to know to ancestors of each assertion — that implicitly gives us the graph of the proof-structure. We will use it for the backtracks of the answers. The function call_ctlstar is modified consequently to take into account the establishment of this field .pred.

The difficulty in this parallel case is especially the management of the sends. Indeed, we do not know, *a priori*, the answer of a sent assertion — this case appear when we compute the validity of $s \vDash \varphi$. Thus, we need to modify the backtracking when an answer is unknown. We thus consider a third possibility of answer: $\bot$ (and the following equation $\neg\bot = \bot$) for the case where we cannot conclude. An assertion which does not already know its answer will have its field .flag equal to $\bot$. Note that all assertions have their fields .flag initialised at $\bot$.

(d) Other methods

We modify the functions call_ctlstar and up_ctlstar accordingly by the adding of an additional field for each disjunctive and conjunctive assertion: .wait — see Figure 3.26. Initially .wait is a set containing the two children of the assertion, like the field .children. If the children of a conjunctive or disjunctive assertion return an answer equal to $\bot$, *i.e.* each one does not known their answers, then the child assertion will be remove of the field .children but retained in the field .wait so that we know this assertion has not its answer. This trick allow us to conclude on the answer of the ancestor assertion, answer possibly equal to $\bot$.

Take for example the assertion $\sigma \vdash \varphi_1 \vee \varphi_2$ which has for children $\sigma_1 \vdash \varphi_1$ and $\sigma_2 \vdash \varphi_1$. Initially, σ.children $= \sigma$.wait $= \sigma_1, \sigma_2$. Suppose that σ calls σ_1. σ_1 is then removed of the field

```
1  def call_ctlstar(σ) is
2    if σ.V
3      return σ.flag
4    else
5      σ.V ←True
6      case σ
7        s ⊢ p where p ∈ {a, ¬a}, a ∈ A :
8          σ.flag ←s ⊨ p
9          ret_ctlstar(σ)
10       s ⊢ φ1 ∧ φ2 :
11         σ1 ←s ⊢ φ1
12         σ2 ←s ⊢ φ2
13         σ1.pred ←σ1.pred∪{σ}
14         σ2.pred ←σ2.pred∪{σ}
15         σ.wait ←σ.children ←{σ1, σ2}
16         loop_ctlstar(σ)
17       s ⊢ φ1 ∨ φ2 :
18         σ1 ←s ⊢ φ1
19         σ2 ←s ⊢ φ2
20         σ1.pred ←σ1.pred∪{σ}
21         σ2.pred ←σ2.pred∪{σ}
22         σ.wait ←σ.children ←{s ⊢ φ1, s ⊢ φ2}
23         loop_ctlstar(σ)
24       s ⊢ A(φ) :
25         call_ltl(σ)
26       s ⊢ E(φ) :
27         σ1 ←s ⊢neg(Eφ)
28         σ1.pred ←σ1.pred∪{σ}
29         σ.children ←{σ1}
30         loop_ctlstar(σ)

1  def loop_ctlstar(σ) is
2    if σ.children ≠ ∅
3      child ←σ.children.pop()
4      child.parentCTL∗ ←σ
5      todo.push(child)
6    else
7      ret_ctlstar(σ)

1  def ret_ctlstar(σ) is
2    if σ.parentCTL∗ ≠ ⊥
3      up_ctl∗(σ.parentCTL∗, σ)
4    elif σ.parentLTL ≠ ⊥
5      ret_ltl(σ)
6    else
7      ret_trace(σ)
```

```
1  def up_ctlstar(σ,child) is
2    case σ
3      s ⊢ φ1 ∧ φ2 :
4        if child.flag = True
5          σ.wait.pop(child)
6          if σ.wait = ∅
7            σ.flag = True
8            ret_ctlstar(σ)
9          else
10           loop_ctlstar(σ)
11       elif child.flag = False
12         σ.wait = ∅
13         σ.flag ←False
14         ret_ctlstar(σ)
15       else
16         if σ.children = ∅
17           σ.flag = ⊥
18           ret_ctlstar(σ)
19         else
20           loop_ctlstar(σ)
21     s ⊢ φ1 ∨ φ2 :
22       if child.flag = True
23         σ.wait = ∅
24         σ.flag ←True
25         ret_ctlstar(σ)
26       elif σ.flag ←False
27         σ.wait.pop(child)
28         if σ.wait = ∅
29           σ.flag = False
30           ret_ctlstar(σ)
31         else
32           loop_ctlstar(σ)
33       else
34         if σ.children = ∅
35           σ.flag = ⊥
36           ret_ctlstar(σ)
37         else
38           loop_ctlstar(σ)
39     s ⊢ Aφ :
40       σ.flag ←flag
41       ret_ctlstar(σ)
42     s ⊢ Eφ :
43       σ.flag = not child.flag
44       ret_ctlstar(σ)
```

Figure 3.26. CTL* decomposition part for the CTL* model-checking algorithm.

.children of σ but is retained in the field .wait. The field .wait will contain the children assertions which already do not know their answers. After some computations, σ_1 returns its answer, suppose $\bot$. Therefore we can not conclude the answer of σ. Suppose now that σ now calls σ_2. σ_1 is thus removed of the field .children of σ. After some computations, σ_2 returns its answer, suppose True. σ_2 is removed of the field .wait, because its answer is now known. But the field .wait of σ, containing σ_1 ensures us that we can not conclude.

The function up_ctlstar is also modified to take into account these new cases.

(e) Management of backtracking the answer

The backtrack of the answers is problematic, because the functions that manage the backtracking, namely ret_ltl, up_ltl, ret_ctlstar and up_ctlstar, have to manage a backtrack of the answers

```
1 def sweep(rcv) is
2   for σ in CACHE
3     if σ.flag ≠ ⊥
4       dump σ
5     else
6       delete σ {remove from mem,CACHE,pred,parent}
7   CACHE.update(rcv)
```

Figure 3.27. Sweep-line procedure for the Algorithm for parallel computing of the CTL* model checking.

with matching the backtrack of the function's calls. That is to say that the backtrack of the answer of an assertion σ to its father σ' coincides with the end of treatment of its call.

If σ' is the father's link (in the proof-structure or in the global graph connecting the decomposition of the formulae which does not begin by A and the different LTL proof-structures, launched from the formulae beginning by A *i.e.* sessions) then a father's call, in the sense where σ' can call the treatment of σ.

The functions up_ltl and up_ctlstar call respectively the functions loop_ltl and loop_ctlstar which continue the exploration on the child remaining with for the function loop_ltl, eventually an unstacking of the LTL exploration's stack namely stack. These functions are defined un Figure 3.29.

The connexions of backtracking between LTL sessions and CTL* sessions performed *via* the functions ret_ltl and ret_ctlstar whose can recursively call each other following that the field .parentCTL∗ or .parentLTL is at $\perp$ or not.

To take into account the backtracks of answers which do not match the backtracks of call's functions we consider the function up_trace modelled on up_ctlstar wich will manage the backtrack of answer on the decomposition of formula, having been made in the cases where the formulae does not begin by A, together with ret_trace which will manage in particular the backtracking on the LTL arc. These functions are defined in Figure 3.28.

Remark that each sent assertion has its fields .parentLTL and .parentCTL∗ at $\perp$. And these assertions are not called by their father, in the sense where their father does not put them in the stack of assertion awaiting of exploration namely todo.

In the following, we note: local(σ) if and only if cpu(σ) = my_pid. For the management of the sends, we use the sets snd_todo and snd_back using to store the assertion to send for, respectively, continue the exploration and to backtrack their answers. The sets rcv and back, are respectively, the set of assertions to explore and whose answers are to backtrack. The variable **done** is True when the initial assertion has its answer.

(f) Sweep line technical

We recall that states and thus assertions do not overlap between different slices — see the chapter about the state-space computations. But this does not still work since some assertions do not have their answer (equal to $\perp$). We can thus not sweep them into disks when changing of slice. To continue to sweep assertions that are no longer needed (they have their answer and are belong to a previous slice), we used a variable CACHE which contains all the assertions — we recall that the implicit graph for proof-structures and LTL sessions is memorised by the .pred field. At each end of super-step, we iterate on this CACHE to sweep into disk the unnecessary assertions – see Figure 3.26

(g) Examples

Considering the LTS whose the only nodes are s and s' the arcs $s \rightarrow_R s'$ and $s' \rightarrow_L s'$ and the hypothesis $s' \not\models p$ and $s' \models q$. We also consider: cpu(s) $\neq$ cpu(s'). We want to check that

```
1  def up_trace(σ,child) is
2    case σ
3      s ⊢ φ1 ∧ φ2 :
4        if child.flag = True
5          σ.wait.pop(child)
6          if σ.wait = ∅
7            σ.flag = True
8            ret_trace(σ)
9        elif child.flag = False
10         σ.wait = ∅
11         σ.flag ←False
12         ret_trace(σ)
13       else
14         if σ.wait = ∅
15           σ.flag = ⊥
16           ret_trace(σ)
17     s ⊢ φ1 ∨ φ2 :
18       if child.flag = True
19         σ.wait = ∅
20         σ.flag ←True
21         ret_trace(σ)
22       elif σ.flag ←False
23         σ.wait.pop(child)
24         if σ.wait = ∅
25           σ.flag = False
26           ret_trace(σ)
27       else
28         if σ.wait = ∅
29           σ.flag = ⊥
30           ret_ctlstar(σ)
31     s ⊢ Aφ :
32       σ.flag ←flag
33       ret_trace(σ)
34     s ⊢ Eφ :
35       σ.flag = not child.flag
36       ret_trace(σ)

1  def ret_trace(σ) is
2    if not local(σ) then snd_back ←snd_back ∪ {σ}
3    for child in σ.pred
4      up_trace(σ,child)
```

Figure 3.28. Additional backtrack procedure for the Algorithm for parallel computing of the CTL* model checking.

$\sigma \vDash A(Xp) \vee E(Xp)$.

Let us note $\sigma_0 \equiv s \vdash A(Xp) \vee E(Xq)$, $\sigma_1 \equiv s \vdash A(Xp)$, $\sigma_2 \equiv s \vdash E(Xq)$, $\sigma'_1 \equiv s \vdash A(p)$, $\sigma'_2 \equiv s \vdash A(X\neg q)$ and $\sigma''_2 \equiv s \vdash A(\neg q)$.

The following shows the running of the algorithm for one single machine to apprehend intuitively the operations of the algorithm:

```
1  par_modchkCTL*(σ0)
2
3    -- SUPER STEP 1 --
4    todo = [σ0]
5    call_ctlstar(σ0)
6      σ0.wait = {σ1, σ2}
7      σ0.children = {σ1, σ2}
8      loop_ctlstar(σ0)
9        σ0.children = {σ2}
10       todo = [σ1]
11   call_ctlstar(σ1)
12     call_ltl(σ1)
13       stack = [σ1]
14       σ1.flag = ⊥ and send={σ'1}
15       ret_ltl(σ1)
16         stack = []
17         ret_ctlstar(σ1)
18           up_ctlstar(σ0, σ1, ⊥)
19             loop_ctlstar(σ0)
20               σ0.children = ∅
21               todo = [σ2]
22   call_ctlstar(σ2)
23     σ2.children = {σ'2}
24     loop_ctlstar(σ2)
25       σ2.children = ∅
26       todo = [σ'2]
27   call_ctlstar(σ'2)
28       call_ltl(σ'2)
29       stack = [σ'2]
30       σ'2.flag = ⊥ and send={σ'1, σ''2}
31       ret_ltl(σ'2)
32         stack = []
33         ret_ctlstar(σ'2)
34           up_ctlstar(σ2, σ'2, ⊥)
35             σ2.flag = ⊥
36             ret_ctlstar(σ2)
37               up_ctlstar(σ0, σ2, σ2.flag)
38                 ret_ctlstar(σ0)
39
40   -- SUPER STEP 2 --
41   todo = [σ'1, σ''2]
42   call_ctlstar(σ'1)
43     call_ltl(σ'1)
44       stack = [σ'1]
45       σ'1.flag = False
46       ret_ltl(σ'1)
47         stack = []
48         ret_ctlstar(σ'1)
49           up_ctlstar(σ1, σ'1, False)
50             σ1.flag ←False
51             ret_ctlstar(σ1)
52               snd_back ←snd_back ∪ {σ1}
53
54   todo = [σ''2]
55   call_ctlstar(σ''2)
56     call_ltl(σ''2)
57       stack = [σ''2]
58       σ'1.flag = False
59       ret_ltl(σ''2)
60         stack = []
```

```
61          ret_ctlstar(σ''_2)
62            up_ctlstar(σ'_2, σ''_2)
63              σ'_2.flag ←False
64              ret_ctlstar(σ'_2)
65                snd_back ←snd_back ∪ {σ'_2}
66
67  −− SUPER STEP 3 −−
68  back = [σ_1, σ'_2]
69  ret_trace(σ_1)
70    up_trace(σ_0, σ_1)
71  back = [σ'_2]
72    up_trace(σ_2, σ'_2)
73      σ_2.flag ←True
74      ret_trace(σ_2)
75      up_trace(σ_0, σ_2)
76      σ_0.flag ←True
```

Now, we want to check that $\sigma \equiv s \vDash A(Xp \vee A(Xq))$.

Let us note $\sigma_0 \equiv s \vdash A(Xp \vee A(Xq))$, $\sigma_1 \equiv s \vdash A(Xp)$, $\sigma_2 \equiv s \vdash A(A(Xq))$, $\sigma'_1 \equiv s' \vdash A(p)$, $\sigma'_2 \equiv s \vdash A(Xq)$ and $\sigma''_2 \equiv s' \vdash A(q)$.

The following shows the running of the algorithm for one single machine to apprehend intuitively the operations of the algorithm:

```
1  par_modchkCTL*(σ_0)
2
3    −− SUPER STEP 1 −−
4    todo = [σ_0]
5    call_ctlstar (σ_0)
6      σ_0.wait = {σ_1, σ_2}
7      σ_0.children = {σ_1, σ_2}
8      loop_ctlstar(σ_0)
9        σ_0.children = {σ_2}
10       todo = [σ_1]
11   call_ctlstar (σ_1)
12     call_ltl (σ_1)
13       stack = [σ_1]
14       σ_1.flag = ⊥ and send={σ'_1}
15       ret_ltl (σ_1)
16         stack = []
17         ret_ctlstar (σ_1)
18           up_ctlstar(σ_0, σ_1)
19             loop_ctlstar(σ_0)
20               σ_0.children = ∅
21               todo = [σ_2]
22   call_ctlstar (σ_2)
23     call_ltl (σ_2)
24       σ_2.children = {σ'_2}
25         loop_ltl(σ_2)
26         σ_2.children = ∅
27         todo = [σ'_2]
28   call_ctlstar (σ'_2)
29       call_ltl (σ'_2)
30       stack = [σ_2, σ'_2]
31       σ'_2.flag = ⊥ and send={σ'_1, σ''_2}
32       ret_ltl (σ_2, σ'_2)
33         σ_2.flag = ⊥
34         loop_ltl(σ_2)
35         stack = ϵ
36         ret_ltl (σ_2)
37           ret_ctlstar (σ_2)
38             up_ctlstar(σ_0, σ_2)
39               σ_0.flag = ⊥
40                 ret_ctlstar (σ_0)
41
42   −− SUPER STEP 2 −−
43   todo = [σ''_2, σ'_1]
44   call_ctlstar (σ'_1)
45     call_ltl (σ'_1)
46       stack = [σ'_1]
47       σ'_1.flag = False
48       ret_ltl (σ'_1)
49         stack = []
50         ret_ctlstar (σ'_1)
51           ret_trace(σ'_1)
52             snd_back = {(σ_1, σ'_1)}
53
54   todo = [σ''_2]
55   call_ctlstar (σ''_2)
56     call_ltl (σ''_2)
57       stack = [σ''_2]
58       σ'_2.flag = True
59       ret_ltl (σ''_2)
60         stack = []
61         ret_ctlstar (σ''_2)
62           ret_trace(σ''_2)
63             snd_back = {(σ_1, σ'_1), (σ'_2, σ''_2)}
64
65   −− SUPER STEP 3 −−
66   back = [(σ'_2, σ''_2), (σ_1, σ'_1)]
67   up_trace(σ_1, σ'_1)
68     σ_1.flag = False
69     ret_trace(σ_1)
70       up_trace(σ_0, σ_1)
71
72   back = [(σ'_2, σ''_2)]
73   up_trace(σ'_2, σ''_2)
74     σ'_2.flag = True
75     ret_trace(σ'_2)
76       up_trace(σ_2, σ'_2)
77         σ_2.flag = True
78         ret_trace(σ_2)
79           up_trace(σ_0, σ_2)
80             σ_0.flag = True
```

Figure 3.30 illustrates the progress of our parallel algorithm for CTL* checking. The development of a "new session" (in grey) means that we initiate the generation (in a parallel way) of a proof graph for the checking of an assertion where the formula begin by a "forall" A. During

```
1  def init(σ,valid) is
2    dfn ←dfn+1
3    σ.dfsn ←σ.low ←dfn
4    σ.valid ←{⟨φ1Rφ2,sp⟩ | φ2 ∉ σ
5      ∧(φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)
6      ∧ sp=(sp' if ⟨φ1Rφ2,sp'⟩ ∈ valid, dfn otherwise}

1  def call_ltl(σ) is
2    # init
3    if σ.parentLTL = ⊥
4      valid ←∅
5    else
6      valid ←σ.parentLTL.valid
7    init(σ,valid)
8    σ.V ←True
9    σ.instack ←True
10   stack.push(σ)
11   # start dfs
12   σ.children ←subgoals(σ)
13   case σ.children
14     {True} :
15       σ.flag ←True
16       ret_ltl(σ)
17     {⊥} :
18       σ.flag ←⊥
19       ret_ltl(σ)
20     ∅ :
21       σ.flag ←False
22       ret_ltl(σ)
23     otherwise :
24       loop_ltl(σ)

1  def loop_ltl(σ) is
2    while σ.children ≠ ∅ and σ.flag != False
3      σ' ←σ.children.pick()
4      if σ'.V
5        if σ'.flag = False
6          σ.flag ←False
7        elif σ'.instack
8          σ.low ←min(σ.low, σ'.low, σ'.dfsn)
9          σ.valid ←{⟨φ1Rφ2,sp⟩ ∈ σ.valid | sp≤ σ'.dfsn}
10         if σ.valid = ∅
11           σ.flag ←False
12     else
13       # flag = dfs(σ', σ.valid)
14       σ'.parentLTL ←σ
15       todo.push(σ')
16       return
17   if σ.dfsn = σ.low
18     var top ←⊥
19     while top ≠ σ
20       top ←stack.pop()
21       top.instack ←False
22       if not σ.flag
23         top.flag ←False
24   ret_ltl(σ)

1  def ret_ltl(σ) is
2    if σ.parentLTL ≠ ⊥
3      up_ltl(σ.parentLTL, σ)
4    else
5      stack.pop() (if stack ≠ ∅)
6      ret_ctlstar(σ)

1  def up_ltl(σ,σ') is
2    # flag = dfs(σ',σ.valid)
3    σ.flag ←σ'.flag
4    if σ'.low ≤ σ.dfsn
5      σ.low ←min(σ.low, σ'.low, σ'.dfsn)
6      σ.valid ←σ'.valid
7    loop_ltl(σ)
```

Figure 3.29. LTL part for the Algorithm for parallel computing of the CTL* model checking.

the exploration of this proof graph, when we encounter a subgoal beginning by a "forall" A, we don't make any pause (like in the naive version) of the ongoing explored session. We start other sessions together with the current session in exploration (which increases the parallelism). Furthermore the Figure shows that (like for the naive version) the development of the initial assertion is not necessarily a development of a proof graph because the subgoal of the initial assertion does not begin necessarily by a "forall".

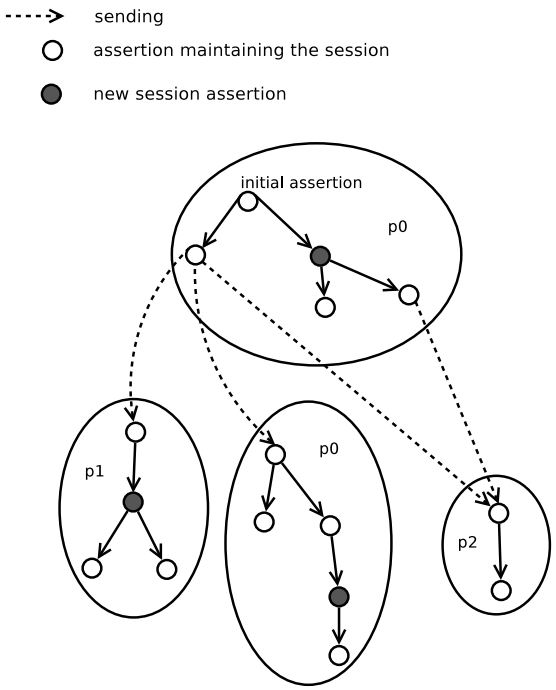


Figure 3.30. Illustration for the parallel algorithm for CTL* checking.

4 Case study

This chapter extends the works of [104, 106].

Contents

4.1 Specification of some security protocols using ABCD 95
4.1.1 Modelisation of the security protocols 95
4.1.2 Full example: the Needham-Schroeder protocol 99
4.1.3 Other examples of protocols 102
4.2 Implementation of the algorithms 105
4.2.1 BSP programming in Python 105
4.2.2 SNAKES toolkit and syntactic layers 110
4.2.3 Parallel algorithms 113
4.3 State space generation's benchmarks 117
4.4 LTL and CTL*'s benchmarks 119

This chapter concerns the practical part of our work. In a first time, we present the specification of security Protocols by the langage ABCD and we give several examples of protocols with their modelisation in this langage. Then, we describe the important technologies we use to implement our algorithms: the BSP Python Programming library and the SNAKES toolkit and syntactic layers wich is a Python library to define, manipulate and execute coloured Petri nets [178]. Then we give the features of the implementation of our parallel algorithms and at last the benchmarks on our differents algoritms.

4.1 Specification of some security protocols using ABCD

In this section, we show how the ABCD language previously introduced can be used to specify and verify security protocols. We consider models of security protocols involving a set of *agents* $\mathcal{A}$ which exchange data (messages) using a network where there is a Dolev-Yaho attacker which is able to read the messages, analyse them with some specific rules and generate new messages over the network. This section is an extension of the work of [181] about security protocols.

4.1.1 Modelisation of the security protocols

(a) Modelling communication and cryptography

Using ABCD, a simple model of a network is a globally shared buffer: to send a message we put its value on the buffer and to receive a message, we get it from the buffer. As explain latter, we actually used two buffers in this document:

buffer snd : **object** = ()
buffer rcv : **object** = ()

$$\frac{\text{if } a \in \mathcal{K}}{\mathcal{K} \vdash a}\ (D0) \quad \frac{\mathcal{K} \vdash \langle a, b\rangle}{\mathcal{K} \vdash a}\ (D1) \quad \frac{\mathcal{K} \vdash \langle a, b\rangle}{\mathcal{K} \vdash b}\ (D2) \quad \frac{\mathcal{K} \vdash b \quad \mathcal{K} \vdash b}{\mathcal{K} \vdash \langle a, b\rangle}\ (D3)$$

$$\frac{\mathcal{K} \vdash k \quad \mathcal{K} \vdash a}{\mathcal{K} \vdash \{a\}k}\ (D4) \quad \frac{\mathcal{K} \vdash \{a\}k \quad \mathcal{K} \vdash k^{-1}}{\mathcal{K} \vdash a}\ (D5)$$

where k and k^{-1} are respectively a key and its inverse.

Figure 4.1. Deductive rules of the Dolev-Yao attacker.

for respectivally sending and receved data for/from agents. These buffers support network communication and allow it to store any token (type object).

Messages can be modelled by tuples and cryptography can be treated symbolically, *i.e.* by writing terms instead of by performing the actual computation. For instance, the first message in the Needham Schroeder protocol, that is agent Alice A sends its None Na to agent Bob B, may be written as a nest of tuples

```
("crypt", ("pub", B), A, Na)
```

where:

- string "crypt" denotes that the message is a cryptogram, the encryption key is thus expected as the second component of the tuple and the following components form the payload;
- tuple ("pub", B) indicates that this is a public key owned by B (we will see later on how to model agents' identities);
- the payload is the message (A, Na) — we will see later on how to model nonces.

Then we need to model agents'identities. In this protocol, we can just use positive integers because no such value is used somewhere else so there is no risk of confusion with another message fragment.

To model nonces, we cannot rely on a random generator unlike in implementations: this would lead to undesired non-determinism and possibly to a quick combinatorial explosion. To correctly model perfect cryptography while limiting state explosion, we must find an encoding such that:

- each nonce is unique;
- a nonce cannot be confused with another value;
- nonces can be generated in a deterministic way.

In our case, a simple solution is to program them a Python class **Nonce**. The constructor expects an agent's identity; for instance, **Nonce**(1) denotes the nonce for the agent whose identity is 1. Equality of two nonces is then implemented as the equality of the agents who own these nonces.

Using this modelling, messages actually travel in plain text over the network. But if we adopt the Dolev&Yao model [77] and correctly implement it, this is a perfectly valid approach.

(b) Modeling the attacker

We consider models of security protocols where a Dolev-Yao attacker [77] resides on the network and which is an specific agent generally called Mallory. An execution of such a model of attacker on the network is thus a series of message exchanges as follows.

1. An agent sends a message on the network.

2. This message is captured by the Dolev-Yao attacker that tries learn from it by recursively decomposing the message or decrypting it when the key to do so is known. Then, the attacker forges all possible messages from newly as well as previously learnt information. Finally, these messages (including the original one) are made available on the network.

3. The agents waiting for a message reception accept some of the messages forged by the attacker, according to the protocol rules.

To respect Dolev&Yao's model, we must forbid an agent to generate a nonce using another agent's identity.

So, the Mallory (spy/attacker) agent can read, remove or even replace any message on the network. Moreover, it can learn from the read messages by decomposing them and looking at their content. However, cryptography is considered to be perfect and so, Mallory cannot decrypt a message if it does not know the key to do so. For instance, if it reads ("crypt", ("pub", B), A, Na) on the network, it is allowed to learn A and Na only if it already knows Bob's private key ("priv", B). To correctly implement Dolev&Yao's model, we shall ensure that no agent can perform any action on an encrypted content knowing the key to do so.

To initialise in our program the Dolev&Yao's attacker, we must import the content of a module dolev_yao that also defines class Nonce. We now explain how it works.

Mallory maintains a knowledge base $\mathcal{K}$ containing all the information learnt so far and repeatedly executes the following algorithm:

1. Get one message m from the network;

2. Learn from m by decomposition or decryption using a key already in $\mathcal{K}$; whenever a new piece of information is discovered, add it to $\mathcal{K}$ and recursively learn from it; this learn is perform by applying the deductive rules of the Figure 4.1; each time a new message not in $\mathcal{K}$ is found, it is add to the knownledge of Mallory; this is apply until no new messages for the knownledge can be deduced;

3. Optionally, compose a message from the information available in k and send it on the network.

The last action is optional, which means that a message may be removed from the network with nothing to replace it. This corresponds to a message destruction, which the attacker is allowed to do. Notice also that, when composing a new message, Mallory may rebuild the message it has just stolen. This corresponds to a message eavesdropping, which the attacker is also allowed to do.

The rules (Figure 4.1) allows the intruder to encrypt any message if it has a key (especially its own public key, rule D4), decompose or recompose messages (rules D1–3), decrypt a message code with a key if it knows the inverse key — rule D5 and in case of a symetric key, we have $k = k^{-1}$ otherwise k and k^{-1} are genrally public and private keys. It is easy to see that the intruder could no decrypt a crypted message it has not the key.

We can also note the deductive Dolev-Yao rules can generate an infinite numbers of messages in the knownledge. For example, with a and b in $\mathcal{K}$, we can deduced $\langle a, b\rangle$, $\langle a, \langle a, b\rangle\rangle$, $\langle a, \langle a, \langle a, b\rangle\rangle\rangle$ and so long. To stay in a bound model and thus in a bound state-space verification, two classicals limitations are imposed to the machineray of the "learn" phase of the attacker:

1. Only generate messages that can be read by honest agents used their types; for example, if the agents can only reads a pair of Nonces, the attacker would only have in its knownledge all possible pair of Nonces that it can deduced from past exchange; note that this reduction can be done at each stage of the protocole (to generated all the time only what the agents can read) or not that is the knownledge grow at its maximum all the time; we have currently choise this solution for implementation convenience;

2. Using what is know to be a "lasy" attacker: the knownledge is built as a set of constraint rules (mainly Horn rules) which reduce its size; for example, in the case of a pair of Nonces, the contraints would be generated in such a way tha only Nonce that can be deduced from the contraints of the knowledge could be accepted; this solution is used in the AVISPA

tool [162] to reduce the state space and thus accelerated the verification; the side effect of this method is that if constraints are sufficiantly generics, a proof of validity for an unbounded number of agents can be extracted.

The hard part in modelling this attacker is the message decomposition and composition machinery. This is feasible with just Petri nets (and has been done in [37]) but is really complicated and leads to many intermediate states that quickly make the state space intractable. Fortunately, using ABCD, we can implement this part in Python. So, module dolev_yao also provides a class Spy that implements Dolev&Yao's attacker. Only the algorithmic part is implemented, taking into account our conventions about symbolic cryptography. For instance, tuples like `("crypt",`$\cdots$`)` or `("pub",`$\cdots$`)` are correctly recognised as special terms.

To reduce combinatorial explosion in the state spaces, an instance of Spy is given a signature of the protocol. This consists in a series of nested tuples in which the elements are either values or types. Each such tuple specifies a set of messages that the protocol can exhibit. For instance, for the past messages we get three types of message:

- `("crypt", ("pub", int), int, Nonce)` corresponding to the first message;
- `("crypt", ("pub", int), Nonce, Nonce)` corresponding to the second message;
- `("crypt", ("pub", int), Nonce)` corresponding to the third message.

This information is exploited by the attacker to avoid composing pieces of information in a way that does not match any possible message (or message part) in the attacked protocol. Without this mechanism, learning just one message m would lead the attacker to build an infinite knowledge containing, *e.g.* (m, m), (m, m, m), $(m, (m, m))$, *etc.* However, this would be completely useless unless such values would be parts of the protocol and may be accepted as a message by an agent if these values were sent on the network. So, the attacker uses the protocol signature to restrict the knowledge to valid messages or message parts.

The knowledge part of the attacker is modelled by a Petri net place *i.e.* by an ABCD buffer. As the main goal of Mallory is to read messages from the network and to send new messages, it reads messages from snd, decompose it and generate new messages from its knowledge which can be the place rcv: this place would be thus all possible messages for normal agents that is "normal one" and possible "attacks". All the messages is the knowledge of the intruder Mallory.

This allows to reduce the size of the markings (there is no a specific place/buffer for the knowledge) and their number during computing the marking graph since their is not intermediate markings of copy the message from the knowledge to the network: both are the same buffer.

This also allows to observe in the state space what the attacker has learnt, for instance to check for leaking secrets. This knowledge has to be initialised, in our case, we would like Mallory to be able to initiate a communication with another agent. So we shall provide her with an identity, and the corresponding nonce and private key. We shall also provide the list of other agents and their public keys. So, Mallory is specified in Figure 4.2 as follow: parameter **this** is like for the other agents, parameter set_sessions is intended to be a tuple of initially known pieces of information that is Mallory's knowledge is declared and initialised; it contains her identity, nonce and private key, plus all the information from set_sessions. An instance of Spy is created in a buffer **spy**, with the signature of the protocol. Then comes the infinite loop to execute the attacker's algorithm:

- A message m is removed from the network with **snd**$-$(m), the content of the knowledge (which we recall is also the received buffer) is flushed to variable k with **rcv**$\gg$ (k) and replaced with all that can be learnt from k, thanks to **rcv**$\ll$ (s.learn(m, k));
- Messages could be them read from **rcv** by agent if and only if it is a valid message, which is checked in the guard of the agent.

Notice that this model of attacker is generic (except possibly for its initial knowledge) and one may safely cop/paste its code to another specification.

```
net Mallory (this, set_sessions) :
    buffer spy : object = Spy(
        (int, int, int, int, Nonce), #1
        (int, int,
         ("crypt", ("secret", int, int),
          int, int, Nonce, ("secret", int, int, Nonce)),
         ("crypt", ("secret", int, int),
          int, int, Nonce, ("secret", int, int, Nonce))), #2
        (int, int,
         ("crypt", ("secret", int, int),
          int, int, Nonce, ("secret", int, int, Nonce)),
         ("crypt", ("secret", int, int, Nonce), Nonce),
         Nonce), #3
        (int, int,
         ("crypt", ("secret", int, int, Nonce), Nonce)) #4
        )
    [rcv≪ ((this,)
           + tuple(range(1, this))
           + tuple(Nonce((this, s)) for s in set_sessions)
           )]
    ⨟ ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.2. Typical code of Mallory.

(c) Defining a scenario

To create a complete ABCD specification, we need to provide a main term. This consists in a composition of instances of the defined agents. By providing this, we create a scenario, *i.e.* a particular situation that can then be analysed. This naturally bounced scenario with a fix number of agents

Using the ABCD compiler, we can create a PNML file from this system. This file can be loaded from a Python program using SNAKES to build the state space and search for a faulty.

4.1.2 Full example: the Needham-Schroeder protocol

As an illustration of the past section, we model Needham&Schroeder's protocol for mutual authentication [163]. This is quite a small example, but fortunately, this protocol allows to show the most important aspects about applying ABCD to security protocols.

(a) A protocol for mutual authentication

The protocol NS involves two agents Alice (A) and Bob (B) who want to mutually authenticate. This is performed through the exchange of three messages as illustrated in Figure 4.3. In this specification, a message m is denoted by $\langle m \rangle$ and a message encrypted by a key k is denoted by $\langle m \rangle_k$ (we use the same notation for secret key and public key encryption). The three steps of the protocol can be understood as follows:

1. Alice sends her identity A to Bob, together with a nonce N_a; the message is encrypted with Bob's public key K_b so that only Bob can read it; N_a thus constitutes a challenge that allows Bob to prove his identity: he is the only one who can read the nonce and send it back to Alice;

2. Bob solves the challenge by sending N_a to Alice, together with another nonce N_b that is a new challenge to authenticate Alice;

3. Alice solves Bob's challenge, which results in mutual authentication.

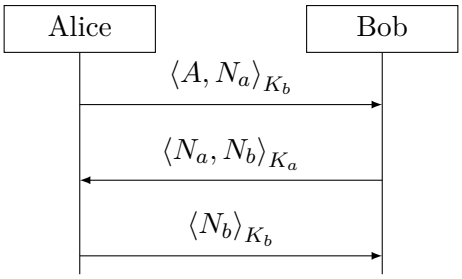


Figure 4.3. An informal specification of NS protocol, where N_a and N_b are nonces and K_a, K_b are the public keys of respectively Alice and Bob.

(b) Known attack

This protocol is well known for being flawed when initiated with a malicious third party Mallory (M). Let us consider the run depicted in Figure 4.4. It involves two parallel sessions, with Mallory participating in both of them.

- When Mallory receives Alice's first message, she decrypts it and forwards to Bob the same message (but encrypted with Bob's key) thus impersonating Alice;
- Bob has no way to know that this message is from Mallory instead of Alice, so he answers exactly as in the previous run;
- Mallory cannot decrypt this message because it is encrypted with Alice's key, but she might use Alice has an oracle and forward the message to her;
- When Alice receives $\langle N_a, N_b \rangle_{K_a}$, she cannot know that this message has been generated by Bob instead of Mallory, and so she believes that this is Mallory's answer to her first message;
- So, Alice sends the last message of her session with Mallory who is now able to retrieve N_b and authenticate with Bob.

In this attack, both sessions (on the left and on the right) are perfectly valid according to the specification of the protocol. The flaw is thus really in the protocol itself, which is called a *logical attack*. This can be easily fixed by adding the identity of the sender to each message (like in the first one), in which case Alice can detect that the message forwarded by Mallory (now it is $\langle B, N_a, N_b \rangle_{K_a}$) is originated from Bob.

(c) Modelisation using ABCD

Figure 4.5 gives a modelisaton of the procotol using ABCD.

Let us consider a simple scenario with one instance of Alice, two of Bob and one of Mallory; a buffer agents will store the identities of Bobs and Mallory so that Alice will contact one or the other at the beginning.

One scenario:

```
buffer agents : int = 2, 3, 4
alice::Alice(1, agents)
| bob::Bob(2)
| bob::Bob(3)
| spy::Mallory(4, ())
```

This scenario includes the possibility for Mallory to try to authenticate with one Bob since we gave to her enough knowledge to play the protocol. So, this simple scenario involves all kind of communications between honest and dishonest agents. Notice that including more than one

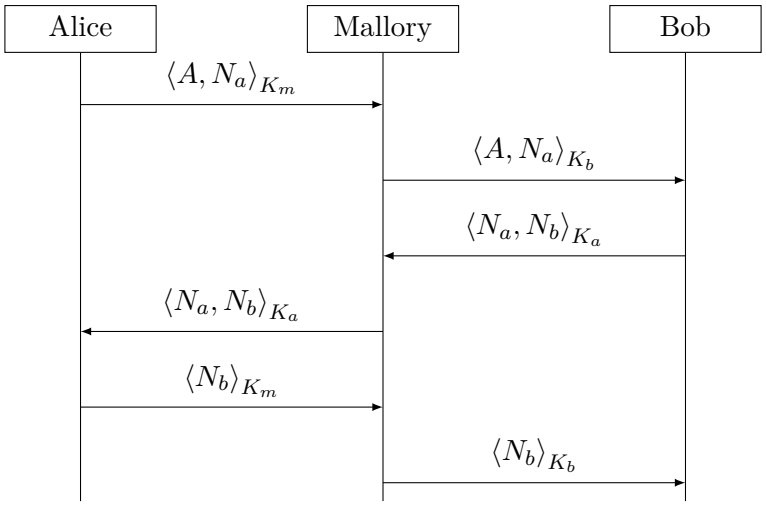


Figure 4.4. An attack on NS protocol where Mallory authenticates as Alice with Bob.

```
net Alice (this, agents) :
    buffer peer : int = ()
    buffer peer_nonce : Nonce = ()
    [agents?(B), peer+(B), snd+("crypt", ("pub", B), this, Nonce(this))]
    ⨾ [rcv?("crypt", ("pub", this), Na, Nb), peer_nonce+(Nb) if Na == Nonce(this)]
    ⨾ [peer?(B), peer_nonce?(Nb), snd+("crypt", ("pub", B), Nb)]

net Bob (this) :
    buffer peer : int = ()
    buffer peer_nonce : Nonce = ()
    [rcv?("crypt", ("pub", this), A, Na), peer+(A), peer_nonce+(Na)]
    ⨾ [peer?(A), peer_nonce?(Na), snd+("crypt", ("pub", A), Na, Nonce(this))]
    ⨾ [rcv?("crypt", ("pub", this), Nb) if Nb == Nonce(this)]

net Mallory (this, init) :
    buffer spy : object = Spy(("crypt", ("pub", int), int, Nonce),
                              ("crypt", ("pub", int), Nonce, Nonce),
                              ("crypt", ("pub", int), Nonce))
    [rcv≪ ((this, Nonce(this), ("priv", this))
          + tuple(range(1, this))
          + tuple(("pub", i) for i in range(1, this))
          + init)]
    ⨾ ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.5. Classical Needham Schroeder protocol in ABCD.

attacker would result in a quick state explosion; fortunately, this is rarely needed and if so, may be simulated by providing more than one identity to the same and only attacker. A possible faulty is here where Alice and Bob are in a final state (*i.e.* their exit places are marked) and mutual authentication is violated (*i.e.* data in buffers peer and peer_nonce of each agent are not consistent).

When build all the possible markings of this scenario, we can shows that Alice authenticated Bob but Bob authenticated Mallory, which corresponds to the known attack. There are also markings showing that both Alice and Bob may authenticate Mallory, but these are just regular runs of two parallel sessions (Alice with Mallory and Bob with Mallory). When looking closer to

the markings, we can see that Mallory is able to use someone else's nonce for authentication: for instance she may use Alice's nonce as a challenge for Alice. This is not an error in the protocol and is fully consistent with the fact that nonces freshness is never tested.

4.1.3 Other examples of protocols

We collect in this section all the protocols being analysed in the document. These experiments would be designed to reveal how well our state-to-processor mapping performs relative to a hand-tuned hash-function, and to determine how various aspects of the new method contribute to the overall performance.

Our cases study were the following protocols which will formally modelise in the next section: the well-known "Kao Chow Authentication v.1", "Otway Rees", "Yahalom", "Woo and Lam". These protocols were found in the Security Protocols Open Repository (SPORE) available at `http://www.lsv.ens-cachan.fr/Software/spore/`.

For all of them, we give an information description and its ABCD specification.

(a) Kao Chow Authentication v.1

The goal of this protocol is the key distribution and authentication using a symmetric keys cryptography with server [139]. This protocol has been designed to prevent the freshness attack on the repeated authentication part of the Neumann Stubblebine protocol. Indeed, in the following, the nonce Na in the ciphers of message 2 prevent a shared key compromised after another run of the protocol to be reused. Figure 4.6 gives a specification of the procotol using ABCD and the protocol can be describe as follow:

```
A, B, S       : principal
Na, Nb        : number
Kab, Kbs, Kas : key

1.    A  ->  S  :     A, B, Na
2.    S  ->  B  :    {A, B, Na, Kab}Kas, {A, B, Na, Kab}Kbs
3.    B  ->  A  :    {A, B, Na, Kab}Kas, {Na}Kab, Nb
4.    A  ->  B  :    {Nb}Kab
```

`Kas` and `Kbs` are symmetric keys whose values are initially known only by `A` and `S`, respectively `B` and `S`. `Na` and `Nb` are nonces for mutual authentication and to verify the authenticity of the fresh symmetric key `Kab`. The messages 3 and 4 are repeated authentication: after that messages 1 and 2 have completed successfully, 3 and 4 can be played several times by `B` before starting a secrete communication with `A` encrypted with the session key `Kab`.

The protocol must guaranty the secrecy of `Kab`: in every session, the value of `Kab` must be known only by the participants playing the roles of `A`, `B` and `S`. When `A`, resp. `B`, receives the key `Kab` in message 3, resp. 2, this key must have been issued in the same session by the server `S` with whom `A` has started to communicate in message 1. The protocol must also ensures mutual authentication of `A` and `B`.

As described in [51], this protocol suffers the same kind of attack as the Denning Sacco freshness attack on Needham Schroeder Symmetric Key, when an older session symmetric key `Kab` has been compromised.

(b) Otway Rees

The goal of this protocol is the distribution of a fresh shared symmetric key between two agents `A` and `B` by trusted server and using symmetric key cryptography with a server [168]. It is assumed that initially `A` and `B` share long term keys `KA` and `KB` with the server, respectively.

Figure 4.7 gives a specification of the procotol using ABCD and the protocol is listed below:

```
A, B, S       :    principal
M, Na, Nb     :    nonce
Kas, Kbs, Kab :    key

1.    A  ->  B  :    M, A, B, {Na, M, A, B}Kas
2.    B  ->  S  :    M, A, B, {Na, M, A, B}Kas , {Nb, M, A, B}Kbs
3.    S  ->  B  :    M, {Na, Kab}Kas, {Nb, Kab}Kbs
4.    B  ->  A  :    M, {Na, Kab}Kas
```

The nonce `M` identifies the session number (a serial number) and provides no security intention, therefore we can safely assume that it is known to all the principals and even the attacker. `Kas` and `Kbs` are symmetric keys whose values are initially known only by `A` and `S`, respectively `B` and `S`. `Kab` is a fresh symmetric key generated by `S` in message 3 and distributed to `B`, directly in message 3, and to `A`, indirectly, when `B` forwards blindly `{Na,Kab}Kas` to `A` in message 4.

The protocol works as follow:

1. `A` generates a fresh nonce `NA`, encrypts it along with the serial number `M` and the names of the principals and sends the encryption as well as the other information to `B`;

2. `B` generates another fresh nonce `NB`, encrypts it with the value `M` and the names of the principals using the shared key and sends it together with what he received to the server `S`;

3. Server `S` generates a fresh session key, `Kas`, encrypts it with the nonces that is known to him after decrypting what he receives, using the long term keys, `KA` and `KB`, respectively; along with the value `M`, the two encryptions are sent to `B`;

4. `B` decrypts the last part of the message he receives using his long term key `KB` and checks whether the nonce `NB` is indeed the one he newly generated and sent out; if this is the case, he then accepts `K` as the new session key and forwards the rest of the message to `A`; `A` also checks the nonce `NA` and decides whether he accepts `K` as the session key.

Now `B` and `A` are able to communicate with each other using the key `K` to encrypt the messages.

The protocol must guaranty the secrecy of `Kab`: in every session, the value of `Kab` must be known only by the participants playing the roles of `A`, `B` and `S`. When `A`, resp. `B`, receives the key `Kab` in message 3, resp. 2, this key must have been issued in the same session by the server `S` with whom `B` has started to communicate in message 2.

There is a claimed attacks [51] which consist of a type flaw, where `A` will accept in last message 4 the triple `(M,A,B)` as a fresh key `Kab`.

(c) Yahalom

The goal of this protocol is the distribution of a fresh symmetric shared key by a trusted server and mutual authentication using symmetric keys and a trusted server [40]. Figure 4.8 gives a specification of the procotol using ABCD and the protocol can be describe as follow:

```
A, B, S       :    principal
Na, Nb        :    number fresh
Kas, Kbs, Kab :    key

A knows :    A, B, S, Kas
B knows :    B, S, Kbs
S knows :    S, A, B, Kas, Kbs
```

```
1.    A  ->  B  :    A, Na
2.    B  ->  S  :    B, {A, Na, Nb}Kbs
3.    S  ->  A  :    {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
4.    A  ->  B  :    {A, Kab}Kbs, {Nb}Kab
```

The fresh symmetric shared key `Kab` is created by the server `S` and sent encrypted, in message 3 both to `A` (directly) and to `B` (indirectly). The protocol must guaranty the secrecy of `Kab`: in every session, the value of `Kab` must be known only by the participants playing the roles of `A`, `B` and `S`. `A` must be also properly authentified to `B`.

A claimed proofs of this protocols is described in [172].

(d) Woo and Lam

This protocol [203] ensures one-way authentication of the initiator of the protocol, `A`, to a responder `B`. The protocol uses symmetric-key cryptography and a trusted third-party server `S`, with whom `A` and `B` share long-term symmetric keys. The protocol uses a fresh and unpredictable nonce `NB` produced by `B`. The protocol narration is listed below, where the keys `KAS` and `KBS` represent the long-term keys that `A` and `B` share with the trusted server `S`. The protocol narration is the following:

```
A, B, S  :    principal
Nb       :    nonce
Kas, Kbs :    skey

1.    A  ->  B  :    A
2.    B  ->  A  :    Nb
3.    A  ->  B  :    {Nb}Kas
4.    B  ->  S  :    {A, {Nb}Kas}Kbs
5.    S  ->  B  :    {Nb}Kbs
```

Figure 4.9 gives a specification of the procotol using ABCD. The Woo-Lam protocol is prone to a type flaw attack by replay.

(e) Wide Mouthed Frog

The goal of this protocol is the distribution of a fresh shared symmetric key between two agents `A` and `B` by trusted server, using symmetric key cryptography with a server and in accordance with timestamps. It is assumed that initially `A` and `B` share long term keys `Kas` and `Kbs` with the server, respectively.

The protocol narration is the following:

```
A, S :    principal
Kas, Kbs, Kab :    symkey
Ta, Ts :    timestamp

1.    A  ->  S  :    A, {Ta, B, Kab}Kas
2.    S  ->  B  :    {Ts, A, Kab}Kbs
```

`A` sends an encrypted message by `Kas` to `S` consisting of the new session key `Kab` with a timestamp `Ta`İf the message is timely, `S` forwards the key `Kab` to `B` by an encrypted message by `Kbs` including the key to share `Kab` with the own timestamp of the server `Ts`Finally, `B` accepts the new key `Kab` if the timestamp `Ts` is later than any other it has received from `S` .

(f) Andrew Secure RPC

The goal of this protocol is the distribution of a fresh shared symmetric key between two agents A and B using symmetric key cryptography where it is assumed that initially A and B share long term keys Kab between them.

The protocol narration is the following:

```
A, B :    principal
Kab, K'ab :    symkey
Na, Nb, N'b :    nonce
succ :    nonce -> nonce

1.    A  ->  B  :    A, {Na}Kab
2.    B  ->  A  :    {succNa, Nb}Kab
3.    A  ->  B  :    {succNb}Kab
4.    B  ->  A  :    {K'ab, N'b}Kab
```

A generates a fresh nonce Na, encrypts it along with the current session key Kab and sends the encryption as well as its own id A to B. B generates a fresh nonce Nb, encrypts it with the value succ(Na) wich is the successor of the nonce Na, using the current session key. After reception, A sends to B the value succ(Nb) encrypted by the session key. Finally, B generates another fresh nonce N'b and the new symmetric key K'ab and send to A these new information encrypted by the current session key Kab. The nonce N'b is intend to be used in a future session.

As described in [40], because of the message 4 contains nothing that A knows to be fresh, this protocol suffers of an attack based on the replay of this message in another session of the protocol to convinced A to accept an old compromised key.

4.2 Implementation of the algorithms

4.2.1 BSP programming in Python

(a) Advantage of Python for parallel programming

The Python language is a famous and general high-level scripting language (mostly object-oriented) which does not need to be presented in much detail here and has only recently become popular in scientific computing.

This language is interactive and interpreted (no compilation/linking). It is thus not for efficient code generation by a compiler (even if run-time code generation is possible) but were designed for convenient programming for fast program test/debug/modify cycle: easy-to-use high-level data types, *e.g.*, nested, heterogeneous list and hash structures, wide file handling functionality, automatic memory management, no declaration of variables or function arguments and extensive run-time testing and clear error messages

Most scientists did not consider Python's programs sufficiently fast for number crunching. However, what made this language interesting in such applications was the idea of multiple language coding: the usually small parts of the code in which most of the CPU time is spent are written in a compiled language, usually Fortran, C, or C++, whereas the bulk of the code can be written in a high-level language. And Python became popular due to its very good integration facilities for compiled languages. For example, the module "Numerical" which implementing efficient array operations for Python, have added significantly to Python's popularity among scientists.

It can be the same thing for parallel programming: programs with a relatively simple communication structure can be implemented with all the communication in Python. However, nothing prevents C or Fortran modules from doing communication as well. Thus, the feature that makes Python particularly suited for high-level parallel programming is the availability of a univer-

sal object serialization mechanism provided by the module "pickle" (and its C implementation "cPickle"). It works so that "pickle.dumps(obj)" returns a string that fully describes the object obj, and the function "pickle.loads(str)" takes such a string and returns a copy of the original object.

Although originally designed for the storage of arbitrary Python objects in files and databases, the pickle module also provides a simple mechanism for sending around arbitrary objects over network connections. However, nothing prevents C or Fortran modules from doing communication of complex objects (*e.g.* needed for state space) as well.

(b) Begenning a BSP computation in Python

Using BSP/Python is done by calling the module in a Python program using:

```
1 from Scientific.BSP import ParData, ParFunction, ParMessages
```

Python itself has no provisions for inter-processor communication or synchronization, a BSP module for Python have therefore be implemented and currently rely on some other library for low-level communication that is MPI (via the Python MPI interface in Scientific Python[16]) and BSPlib [126] — via the BSPlib interface in Scientific Python. The choice between the two is made at runtime, application programmers use only the Python/BSP API in their programs.

At the origin, as BSML [151], Python/BSP program is to be read as a program for a single parallel machine with $\mathbf{p}$ processors in contrast to a MPI program (as well as a C program using BSPlib) which is for a single processor that communicates with $\mathbf{p}-1$ other processors. We will show that this feature can be easily circumvented.

In message-passing programs, communication is specified in terms of local send and receive operations. A Python/BSP program has two levels, local (single processor) and global (all processors) and communications are a synchronized global operation in which all processors participate — as a collective operation in MPI.

In theory, as the parallel vector in BSML, the most important concept for BSP programming in Python is the distinction between local and global objects. Local objects are standard Python objects, they exist on a single processor. Global objects exist on the parallel machine as a whole. They have a local value on each processor, which may or may not be the same everywhere. There are several ways to create global object, corresponding to their typical uses. In the simplest form ParConstant, a global object represents a constant that is available on all processors. Another common situation is that the local representation is a function of the processor number and the total number of processors ParData. Functions being objects in Python, the same distinction between the global and local level applies to functions as well. Python's functions are local functions: their arguments are local objects, and their return values are local objects as well. Global functions take global objects as arguments and return global objects. A global function is defined by one or more local functions that act on the local values of the global objects.

Classicaly, each processor receives an identifier (a number id) between 0 and $\mathbf{p}-1$. All processors are considered equal except for operations that give a special role to one of the processors, this role is by default assigned to processor number 0. The pid and $\mathbf{p}$ could be obtained using:

```
1 def cpu (pid, nprocs) :
2     return pid, nprocs
3
4 pid, nprocs = (x.value for x in ParData(cpu))
```

To not have to manage between local and global objects[1] and writting our programs as BSPlib ones (SPMD ones) but in Python, the solution to circumvent this fact is to make the main function of the Python program as global, that is:

```
@ParFunction
def main (infile) :
    #parallel code
    #main loop of the code

#call this functon with the first argument of the shell
main(sys.argv[1])
```

Now, we can still uses the BSP/Python facilities for communicated Python's objects in a BSP-SPMD fashion. Note that it is not appropriate to follow the example of BSPlib and define separate API routines for sending data and for synchronization, which implies reception. Such a separation would invite erroneous situations in which a routine sends data and then calls another function or method that happens to perform a synchronization. This risk is eliminated in Python/BSP by making synchronization an integral part of every communication operation. A single API call sends data and returns the received data after the synchronization.

(c) BSP-Python's communication routines

According to the BSP model, all communication takes place at the end of a superstep, after the local computations. A superstep is thus simply everything that happens between two communication operations, which in general involves code in several functions and methods.

Python/BSP communication operations are defined as methods on global objects. An immediate consequence is that no communication is possible within local functions or methods of local classes. However, communication is possible within the methods of global classes of functions, which define distributed data types. This is the case for our main global function.

In one important aspect, as in BSML, Python/BSP is much higher-level than BSPlib for C: communication operations can transmit almost any kind of data[2].

BSP/Python propose a set of communication patterns implemented as methods in all of the global data classes. For example, we have:

- put(pid list) which sends the local value to all processors in pid list (a global object whose local value is a list of processor identifiers); returns a global data object whose local value is a list of all the values received from other processors, in unspecified order;
- fullExchange() which sends the local value of each processor to all other processors; returns a global data object whose local value is a list of all the received values, in unspecified order;
- accumulate(operator, zero) which performs an accumulation with operator over the local values of all processors using zero as initial value; the result is a global data object whose local value on each processor is the reduction of the values from all processors with lower or equal number.

In the communication operations described until now, it is always the local value of the global data type that is sent, whether to one or to several receiving processors. In some situations, it is necessary to send different values to different processors. This can in principle be achieved

[1]This model of programming were design to ensure safety be fordib deadlocks: synchronisations would be global and thus do not depend of local values without as being firts globally exchange.

[2]This is achieved by using the general serialization of Python, which generates a unique byte string representation of any data object and also permits the reconstruction of the object from the byte string.

by a series of put operations, but a special communication operation is both more efficient and allows more readable code.

For this purpose, Python/BSP provides a specialized global data type called ParMessages. Its local values are lists (or sets) of data - processor identifier pairs. The method exchange() sends each data item to the corresponding processor and returns another ParMessages object storing the received data items.

This is the method we used in our programsand and is as the all-to-allv of MPI. For example of use, if send is the list (or sets) is data-processor id pairs then

```
recv = ParMessages(send).exchange().value
```

will perform exchange of values and synchronization by sending all the values containing in send. Now, received values are stored in the list recv in an unspecified order and each processor can easally iterate on it.

By make global the main function and using total exchange, we do not used the two levels of BSP/Python and its good way of programming: our programs can thus make deadlocks if one (or more) processors do not participe to the global/collective exchange, *e.g.* no have the same number of super-steps:

```
if pid==0:
   #pure sequential code
else:
   recv = ParMessages(send).exchange().value
   #pure sequential code
```

This will require us to manage the exact same number of super-steps one each processor — which will be easy to do in our case. We have willingly choose this lack of safety to have a more common way of programming and to easally translate the code to more efficient language/libraries (C+MPI) and mainly for more classical tools for model-checking.

(d) Examples of BSP-Python programs

We present here some examples using BSP-Python. We only used the patterns

ParMessages(send).exchange().value

and the fact that the main function has been made global.

Total exchange. One particulary interesting patterns of communication is the total exchange *i.e.* each processor send its local value to other processors and in final, each processor have all those values. This is mainly use in algorithms where we need a global strategy choise for optimise further computations. We can code this function as:

```
def total_exchange(value):
   send=set()
   for i in xranges(nprocs):
      send.add((i,(i,value)))
   rcv=ParMessages(send).exchange().value
   return rcv
```

Note that we can just adding value instead of the the pair (i,value) if knowing from which processor is the value is not necessary — we recall that their is no order of messages using exchange().value. In this case we find the fullExchange() pattern.

The BSP cost would be

$$(\mathbf{p}-1)\times s\times \mathbf{g}+\mathbf{L}$$

where s is the bigger value (in bytes) aims by the processors.

Broadcasting values. Broadcasting a value consist of that a chosen processor send its local value to other processors which could be coded as follow:

```
1 def direct_bcast(sender,value):
2   send=set()
3   if sender==pid:
4     for i in xranges(nprocs):
5       send.add((i,value))
6   rcv=ParMessages(send).exchange().value
7   return rcv.pop()
```

Since each processor received only one value from processor sender, it is thus possible to take the only value in rcv. The BSP cost is:

$$(\mathbf{p}-1)\times\mathcal{S}(v_i)\times\mathbf{g}+\mathbf{L}$$

where $\mathcal{S}(v_i)$ is the size of the broadcasting value v_i.

When $\mathbf{p}$ and $\mathcal{S}(v_i)$ increase, it is clear that this method is not the good one. Another way is the two-phases broadcasting: first, the emitter processor "cut" its value into $\mathbf{p}$ pieces and send each piece to a processor (first super-step); then a total exchange of the received pieces is perform (second super-step); finally each processor "glue" together the received pieces to recompose the initial value. For code it, we also need to "scatter" that is perform the first super-step of the method. The full code would be:

```
1 def two_phase_bcast(sender,value):
2  #scatter
3  if pid==sender:
4     send=cut(value)
5  rcv=ParMessages(send).exchange().value
6  #total echange
7  send.empty()
8  my_piece=rcv.pop()
9  for i in xranges(nprocs):
10    send.add((i,(i,my_piece)))
11 rcv=ParMessages(send).exchange().value
12 #glue
13 return glue(rcv)
```

where we suppose that we have the "cut" function that partition the value into a list of $\mathbf{p}$ pairs (id,piece) and a function "glue" that can aggregate a list of pair (id,piece) into the initial emitted value. The BSP cost would be

$$2\times\frac{(\mathbf{p}-1)\times\mathcal{S}(v_i)}{\mathbf{p}}\times\mathbf{g}+2\times\mathbf{L}+d(v_i)+r(v_i)$$

where $d(v_i)$ is the time to "cut" into $\mathbf{p}$ pieces the initial value v_i of emitter processor and $r(v_i)$ time to pick up the $\mathbf{p}$ pieces.

Parallel Sampling Sort Algorithm. This example is the sampling sort algorithm (PSRS) of Schaeffer in its BSP version [196]. The goal is to have data locally sorted and that processor i have smaller elements than those of processor $i+1$. Data were also need to be well enough balanced. We assume n elements to sort where $\mathbf{p}^3\leq n$ and elements are well distributed over the processors — each processor have $\frac{n}{\mathbf{p}}$ elements.

The PSRS algorithm proceeds as follows. First, the local lists of the processors are sorted independently with a sequential sort algorithm. The problem now consists of merging the $\mathbf{p}$

sorted lists. Each process selects from its list $\mathbf{p}+1$ elements for the primary sample and there is a total exchange of these values. In the second super-step, each process reads the $\mathbf{p}\times(\mathbf{p}+1)$ primary samples, sorts them and selects $\mathbf{p}$ secondary (main) samples. Noted that the main sample is thus the same on each processor. That allows a global choice of how remapping the data. In the third super-step, each processor picks a secondary block and gathers elements that do belong to the assigned secondary block. In order to do this, each processor i sends to processor j all its elements that may intersect with the assigned secondary blocks of processor j.

For simplify we suppose element of the "same size". The BSP cost of the first super-step is thus:

$$\frac{n}{\mathbf{p}}\times\log(\frac{n}{\mathbf{p}})\times c_e+\frac{n}{\mathbf{p}}+(\mathbf{p}\times(\mathbf{p}+1)\times s_e)\times\mathbf{g}+\mathbf{L}$$

where c_e is the time to compare two elements and s_e size of an element. It is easy to see that each processor send at most $\frac{3n}{\mathbf{p}}$ elements. The BSP cost of the second super-step is thus:

$$\frac{n}{\mathbf{p}^2}\times\log(\frac{n}{\mathbf{p}^2})\times c_c+\frac{n}{\mathbf{p}^2}+\frac{3n}{\mathbf{p}}\times s_e\times\mathbf{g}+\mathbf{L}+\text{time}_{\text{fusion}}$$

where the time of merge elements (in a sorting way) is of order of $n/\mathbf{p}$.

Using appropriate functions, that could code as:

```
1  def pssr( lists ):
2   lists .sort()
3   first_sample=select(nprocs,lists)
4   for i in xranges(nprocs):
5       send.add((i,first_sample))
6   rcv=ParMessages(send).exchange().value
7   second_ample=select(nprocs,rcv)
8   send=intervalles(nprocs,second_smaple,lists)
9   rcv=ParMessages(send).exchange().value
10   lists .empty()
11   for x in rcv:
12       lists .add(x)
```

4.2.2 SNAKES toolkit and syntactic layers

SNAKES is a Python library to define, manipulate and execute coloured Petri nets [178]. A large part of the work presented in this document have been implemented within SNAKES or using it.

There exists a wide range of Petri net tools, most of them (if not all) being targeted to a particular variant of Petri nets or a few ones. On the contrary SNAKES provides a general and flexible Petri net library allowing for quick prototyping and development of ad-hoc and test tools using the programming language Python for build the Coloured Petri nets but also Python expression for the colours and the annotations — types and guards.

Python has been chosen as the development language for SNAKES because its high-level features and library allows for quick development and easy maintenance. The choice of Python as a colour domain then became natural since Python programs can evaluate Python code dynamically. Moreover, if Python is suitable to develop a Petri net library, it is likely that it is also suitable for Petri net annotations. It may be added that Python is free software and runs on a very wide range of platforms: this is actually a general requirement as if a software is complicated and works on a very specific platform, it is likely that only few people will use it.

In this section, we will not describe all the SNAKES library but only the needed for this work. We refear to the web site of SNAKES [17] or [178] for more details.

(a) Architecture

SNAKES is centred on a core library that defines classes related to Petri nets. Then, a set of extension modules, *i.e.*, *plugins*, allow to add features to the core library or to change its behaviour. SNAKES is organised as a core hierarchy of modules (plus additional internal ones not listed here):

- snakes is the top-level module and defines exceptions used throughout the library;
- snakes.data defines basic data types (*e.g.* multisets and substitutions) and data manipulation functions (*e.g.* Cartesian product);
- snakes.typing defines a typing system used to restrict the tokens allowed in a place;
- snakes.nets defines all the classes directly related to Petri nets: places, transitions, arcs, nets, markings, reachability graphs, etc; it also exposes all the API from the modules above;
- snakes.plugins is the root for all the extension modules of SNAKES.

SNAKES is designed so that it can represent Petri nets in a very general fashion:

- Each transition has a guard that can be any Python Boolean expression;
- Each place has a type that can be an arbitrary Python Boolean function that is used to accept or refuse tokens;
- Tokens may be arbitrary Python objects;
- Input arcs (*i.e.* from places to transitions) can be labelled by values that can be arbitrary Python object (to consume a known value), variables (to bind a token to a variable name), tuples of such objects (to match structured tokens, with nesting allowed), or multisets of all these objects (to consume several tokens); new kind of arcs may be added (*e.g.* read and flush arcs are provided as simple extensions of existing arcs);
- Output arcs (*i.e.* from transitions to places) can be labelled the same way as input arcs, moreover, they can be labelled by arbitrary Python expressions to compute new values to be produced;
- A Petri net with these annotations is fully executable, the transition rule being that of coloured Petri nets: all the possible enabling bindings of a transition can be computed by SNAKES and used for firing.

SNAKES delegates all the computational aspects of Petri nets to Python. In particular, a token is an arbitrary Python object, transitions execution can be guarded by arbitrary Python Boolean expressions, and so on. As a result, a Petri net in snakes is mainly a skeleton with very general behavioural rules (consume and produce tokens in places through the execution of transitions) and with the full power of a programming language at any point where a computation is required. snakes itself is programmed in Python and uses the capability of the language to dynamically evaluate arbitrary statements. Using the same programming language for SNAKES and its extension language is a major advantage for the generality: Petri nets in snakes can use snakes as a library and work on Petri nets. For instance, as a token in SNAKES may be any Python object, it could be an instance of the Petri net class of SNAKES.

(b) Main features

Apart from the possibility to handle Python-coloured Petri nets, the most noticeable other features of SNAKES used in this work are:

- Flexible typing system for places: a type is understood as a set defined by comprehension; so, each place is equipped with a type checker to test whether a given value can be stored or not in the place; using module snakes.typing, basic types may be defined and complex types may be obtained using various type operations (like union, intersection, difference, complement, etc.); user-defined Boolean functions can also be used as type checkers;

- Variety of arc kinds can be used, in particular: regular arcs, read arcs and flush arcs;
- Support for the Petri net markup language (PNML) [177]: Petri nets may be stored to or loaded from PNML files;
- Fine control of the execution environment of the Python code embedded in a Petri net;
- Flexible plugin system allowing to extend or replace any part of SNAKES;
- SNAKES is shipped with a compiler that reads ABCD specifications to produce PNML files or pictures;
- plugin gv allows to layout and draw Petri nets and marking graphs using GraphViz tool [83].

Naturally, SNAKES also provide a tool that transforms ABCD expressions (with Python expression) into Python-coloured Petri nets. That able to manipulate the ABCD expressionz as a Petri net.

Now we show how using SNAKES for our purpose that is not model problem using Petri nets (and thus build Petri nets using SNAKES) because we use ABCD for this but how execute a Petri net and more precisely how firing marking and obtain the child's markings.

(c) Use cases

First of all, if we want to used SNAKES in our Python program, we must to load the package, load a Petri net from a PNML file and obtain the initial marking. That could be done with:

```
1 import snakes.nets
2 # load PNML
3 net = snakes.nets.loads( infile )
4 # get initial marking
5 init = net.get_marking()
```

Now it is possible to obtain all the transitions and place names:

```
1 #geting the list of all the transition 'names
2 all_trans=[t.name for t in net.transition ()]
3 #getting the list of all the place'names
4 places=[p.name for p in net.place()]
```

And now, obtain the marking's child (the successors) from a fixed marking (the initial one or else) can be coded as:

```
1 # set net's marking
2 net.set_marking(marking)
3 for tname in all_trans :
4     # get transition from its name
5     t = net.transition(tname)
6     #obtained all the modes and iter on them
7     for m in t.modes() :
8       # fire 't', get the new marking and reset marking
9       t. fire (m)
10      new_marking = net.get_marking()
11      net.set_marking(marking)
12      #in the following do something on the new state
```

In this code we have a main loop which iter on all the transition'names of the Petri net. In each loop, we take the transition "t" from its name. In order to get the list of enabling bindings for the transition t, one may use t.modes(). Then, we thus iterate on the possible modes "m" of the transition (from the marking). That allow to fire this transition with the each of the modes

and to get a new marking from the net (fire a transition has a side effet of execution on the net) and we load the initial marking in order to go around again the loop.

In this document we also used some properties of the Petri nets generated by ABCD from security protocols problem. First all, we can easally iterate on the places and transitions of net like this

```
for p in net.place() :
    ...
for t in net.transition() :
    ...
```

and we can have the name of the place (resp. transition) p.name (resp. t.name), p.label("net") (resp. t.label("net")), p.label("name"), the status of the place (p.status) that is statuses indicating their roles (buffer of data or control flow of the processes), t.label("action")

4.2.3 Parallel algorithms

It is easy to see that the code is very simple to read and using Python allows to write the code as a quasi-syntactic matching from our theoretical algorithms. The use of the global exchanges of the BSP communication makes the termination problem of parallel state space construction very simple while complicated algorithms are defined in previous papers [18].

However, we explain briefly some points and tricks of implementation used to encode our algorithms.

(a) State Space generation's implementation

Here, we highlight the Python function of the computation of the successors for the naive algorithm to explore the state space. It uses in particular the library Snakes viewed previously. Notice the global variable allrules which list the set of the transitions of the model.

```
def initialize ( infile ) :
    global net, s0, allrules , places
    (...)
    allrules = [t.name for t in net.transition ()]
    (...)


def succ (s) :
    res = set()
    net.set_marking(s)
    for tname in allrules :
        t = net.transition(tname)
        for m in t.modes() :
            t.fire(m)
            res.add(net.get_marking())
            net.set_marking(s)
    return res
```

Our amelioration on this function consists of two successors functions: one for local transitions and the other for the reception transition whose states fired correspond to the sends states or to unsent states but to explore during the next superstep. To do this it suffices to add an argument named allrules which is no longer, therefore, consider as all the transitions of the model, the body of the function remaining the same. The functions of local successors succL() and of reception successors succR() use the function **succ**() by specifying as argument which transitions must be fired.

```
def succ (s, allrules ) :
  (...)

def succL (s) :
```

```
    return succ(s, noht)

def succR (s) :
    return succ(s, ht)
```

All reception transitions denoted ht and all local transitions denoted noht are found during the loading phase of the Petri net. We add a file having the same name as the *pnml* file with the extension .ht, this file indicates the reception transitions. The set of local transitions being found by the computation of the complementary of these transitions. Similarly, are listed in a file having the extension .dp, the reception places used by the hash function; such a hash as we have seen preserves a certain locality.

```
def initialize ( infile ) :
    global net, s0, allrules , places
    net = snakes.nets.loads( infile )
    s0 = net.get_marking()
    dp = [l.strip() for l in open(infile + ".dp") if l.strip()]
    dp.sort()
    ht = [l.strip() for l in open(infile + ".ht") if l.strip()]
    ht.sort()
    noht = [t.name for t in net.transition() if t.name not in ht]
    noht.sort()

def h (m) :
    return reduce(operator.xor, (hash((p, m[p])) for p in dp if p in m), 0) % nprocs
```

Find the places of the processes and the reception transitions to put in the files with the extensions respective .dp and .ht is an easy task wich can be automated. Take as example the Needham Schroeder protocol (whose a specification is given before it), playing a certain scenario. The file of reception transitions contains only transitions of agents performing a reception.

We recall the scenario:

```
buffer agents : int = 2, 3, 4
alice::Alice(1, agents)
| bob1::Bob(2)
| bob2::Bob(3)
| spy::Mallory(4, ())
```

The file of reception transitions (4.12) includes reception transitions preceded by the name of agents which play them in the scenario.

The file of reception places (4.13) includes the reception places preceded by the name of agents which play them in the scenario, or more exactly the places wich are modified during the receptions and remaining unchanged by firing the other transitions.

We give here the implementation of our parallel state space generation algorithm wich benefits, in addition to the previous improvement, of a statistical calculation phase of the states to send for a better balance of communications, and as we have seen, of calculations also. Note the simplicity of expressiveness of Python language, the great simplicity of the code with the corresponding algorithm.

```
from Scientific.BSP import ParData, ParFunction, ParMessages
import snakes.nets
import bspsnk
from snakes.hashables import *
import operator

def cpu (pid, nprocs) :
    return pid, nprocs

pid, nprocs = (x.value for x in ParData(cpu))

def initialize ( infile ) :
```

```
13      global net, s0, allrules, places
14      net = snakes.nets.loads(infile)
15      s0 = net.get_marking()
16      dp = [l.strip() for l in open(infile + ".dp") if l.strip()]
17      dp.sort()
18      ht = [l.strip() for l in open(infile + ".ht") if l.strip()]
19      ht.sort()
20      noht = [t.name for t in net.transition() if t.name not in ht]
21      noht.sort()
22
23  def h(m):
24      return reduce(operator.xor, (hash((p, m[p])) for p in dp if p in m), 0)
25
26  def succ(s, allrules):
27      res = set()
28      net.set_marking(s)
29      for tname in allrules:
30          t = net.transition(tname)
31          for m in t.modes():
32              t.fire(m)
33              res.add(net.get_marking())
34              net.set_marking(s)
35      return res
36
37  def succL(s):
38      return succ(s, noht)
39
40  def succR(s):
41      return succ(s, ht)
42
43  def successor(known, todo):
44      tosend = collections.defaultdict(set)
45      while todo:
46          s = todo.pop()
47          known.add(state)
48          for s_ in succL(s) - known:
49              todo.add(s_)
50          for s_ in succR(s) - known:
51              tosend[h(s_)].add(s_)
52       return tosend
53
54  def BSP_EXCHANGE(tosend):
55      todo = set(tosend[pid])
56      total = sum(len(tosend[k]) for k in xrange(nprocs))
57      for j, (count, states) in ParMessages((i, (total, tosend[i]))
58                                            for i in xrange(nprocs)
59                                            if i != pid).exchange().value:
60          total += count
61          todo.update(states)
62      return total, todo
63
64  def balance(tosend):
65      histo = collections.defaultdict(int)
66      local = tuple((i, len(states)) for i, states in tosend.iteritems())
67      histo.update(local)
68      for j, l in ParMessages((n, local) for n in xrange(nprocs)
69                              if n != pid).exchange().value:
70          for i, c in l:
71              histo[i] += c
72      pack = [hset() for n in xrange(nprocs)]
73      size = [0] * nprocs
74      for c, i in sorted((c, i) for i, c in histo.iteritems(), reverse=True):
75          # this is not efficient in terms of complexity, but fast in
76          # terms of implementation (C code running on short lists)
77          m = size.index(min(size))
78          pack[m].update(tosend[i])
79          size[m] = len(pack[m])
80      return enumerate(pack)
81
82  def exchange(known, tosend):
83      known.clear()
84      return BSP_EXCHANGE(balance(tosend))
85
```

```
@ParFunction
def main ( infile ) :
     initialize ( infile )
    todo = set()
    total = 1
    known = set()
    if h(s0) == pid :
        todo.add(s0)
    while total>0 :
        tosend = successor(known, todo)
        todo, total = exchange(known, tosend)

main(sys.argv[1])
```

(b) LTL and CTL*'s implementation

Our implementation of our algorithm of LTL checking is done via the object paradigm. A class ModchkLTL is used for the body of the algorithm itself. Verification is done by the method par_exploration() which takes as argument the initial assertion including the initial state and the LTL formula it must verify.

```
class ModchkLTL (object) :
    def __init__ (self) :
        (...)
    def init (self, sigma, valid) :
        (...)
    def dfs (self, sigma, valid, send) :
        (...)
    def par_exploration(self, sigma0) :
        (...)
    def BSP_EXCHANGE (self, tosend, flag) :
        (...)
    def exchange (self, send, flag) :
        (...)
```

```
@ParFunction
def callModchkLTL (sigma) :
    mck = ModchkLTL()
    mck.par_exploration(sigma)
```

The object paradigm is helpful especially for the treatment of formulas. The class Sigma manages the processing of assertion via especially the method subgoals() which implements the rules of subgoals used by the algorithm and defined in [28] taking into account the imperatives of our algorithm by filling the set send of elements to send.

```
class Formula (object) :
    (...)
class Or (Formula) :
    (...)
class Not (Formula) :
    (...)
class And (Formula) :
    (...)
class Forall (Formula) :
    (...)
class Exists (Formula) :
    (...)
class Next (Formula) :
    (...)
class Until (Formula) :
    (...)
class WeakUntil (Formula) :
    (...)
```

```
class Atom (Formula) :
    (...)
class Deadlock (Atom) :
    (...)
class State (object) :
    (...)
class Sigma (object) :
    (...)
    def subgoals (self, send) :
        (...)
        # R1
        if p.s and p(self) :
            (...)
        # R2
        elif p.s :
            (...)
        # R3
        elif isinstance(p, Or) :
            (...)
```

```
    # R4
    elif isinstance(p, And) :
      (...)
    # R5
    elif isinstance(p, Until) :
      (...)
    # R6
    elif isinstance(p, WeakUntil) :
      (...)
    # R7
    elif all(isinstance(x, Next) for x in self.a) :
      (...)
```

Notice that, by postponing communication, this algorithm allows buffered sending and forbids sending several times the same state.

4.3 State space generation's benchmarks

In order to evaluate our algorithm, we have implemented a prototype version in Python, using SNAKES [178] for the Petri net part (which also allowed for a quick modelling of the protocols, including the inference rules of the Dolev-Yao attacker) and a Python BSP library [128] for the BSP routines (which are close to an MPI "alltoall"). We actually used the MPI version (with MPICH) of the BSP-Python library. While largely suboptimal (Python programs are interpreted and there is no optimisation about the representation of the states in SNAKES), this prototype nevertheless allows and accurate *comparison* of the various algorithms.

With respect to the presented algorithms, our implementations differ only on technical details (*e.g.* value *total* returned by **BSP_EXCHANGE** is actually computed by exchanging also the number of values sent by each processor) and minor improvements (*e.g.* we used in-place updating of sets and avoided multiple computations of $\mathsf{cpu}(s)$ using an intermediate variable).

The benchmarks presented below have been performed using a cluster with 16 PCs connected through a Gigabyte Ethernet network. Each PC is equipped with a 2GHz Intel Pentium dual core CPU, with 2GB of physical memory. This allowed to simulate a BSP computer with 32 processors equipped with 1GB of memory each. MPICH were used as low level library for BSP-Python.

These experiments are designed to compare the performances of the two implementations. Our cases study involved the following five protocols: (1) Needham-Schroeder (NS) public key protocol for mutual authentication; (2) Yahalom (Y) key distribution and mutual authentication using a trusted third party; (3) Otway-Rees (OR) key sharing using a trusted third party; (4) Kao-Chow (KC) key distribution and authentication; (5) Woo and Lam Pi (WLP) authentification protocol with public keys and trusted server. These protocols and their security issues are documented at the Security Protocols Open Repository (SPORE[3]).

For each protocol, using ABCD, we have built a modular model allowing for defining various scenarios involving different numbers of each kind of agents — with only one attacker, which is always enough. We note these scenarios NS$-x-y \equiv x$ Alices, y Bobs with one unique sequential session; Y(resp. OR, KC and WLP)$-x-y-z\_n \equiv x$ Servers, y Alices, z Bobs, n sequential sequential sessions.

We give here the total time of computation. We note **SWAP** when at least one processor swaps due to a lack of main memory for storing its part of the state space. We also note **COMM** when this situation happens in communication time: the system is unable to received data since no enough memory is available. We also give the number of states. We have for the Needham-Schroeder protocol:

Scenario	Naive	Balance	Nb_states
NS_1-2	0m50.222s	0m42.095s	7807
NS_1-3	115m46.867s	61m49.369s	530713
NS_2-2	112m10.206s	60m30.954s	456135

For the Yahalom protocol:

[3] http://www.lsv.ens-cachan.fr/Software/spore

Scenario	Naive	Balance	Nb_states
Y_1-3-1	12m44.915s	7m30.977s	399758
Y_1-3-1_2	30m56.180s	14m41.756s	628670
Y_1-3-1_3	481m41.811s	25m54.742s	931598
Y_2-2-1	2m34.602s	2m25.777s	99276
Y_3-2-1	**COMM**	62m56.410s	382695
Y_2-2-2	2m1.774s	1m47.305s	67937

For the Otway-Rees protocol:

Scenario	Naive	Balance	Nb_states
OR_1-1-2	38m32.556s	24m46.386s	12785
OR_1-1-2_2	196m31.329s	119m52.000s	17957
OR_1-1-2_3	411m49.876s	264m54.832s	22218
OR_1-2-1	21m43.700s	9m37.641s	1479

For the Woo and Lam Pi protocol:

Scenario	Naive	Balance	Nb_states
WLP_1-1-1	0m12.422s	0m9.220s	4063
WLP_1-1-1_2	1m15.913s	1m1.850s	84654
WLP_1-1-1_3	**COMM**	24m7.302s	785446
WLP_1-2-1	2m38.285s	1m48.463s	95287
WLP_1-2-1_2	**SWAP**	55m1.360s	946983

For the Kao-Chow protocol:

Scenario	Naive	Balance	Nb_states
KC_1-1-1	4m46.631s	1m15.332s	376
KC_1-1-2	80m57.530s	37m50.530s	1545
KC_1-1-3	716m42.037s	413m37.728s	4178
KC_1-1-1_2	225m13.406s	95m0.693s	1163
KC_1-2-1	268m36.640s	159m28.823s	4825

We can see that the overall performance of our dedicated implementation (call balance) is always very good compared to the naive and general one. This holds for large state spaces as well as for smaller ones. Furthermore, the naive implementation can swap which never happens for the “balance” one.

To see the differences in behaviour (and not only execution time), we show some graphs for several scenarios. In the Figures 4.15–4.18, we have distinguished: the computation time that essentially corresponds to the computations of successor states on each processor (in black); the communication time that corresponds to states exchange and histogram computations (in grey); the waiting times that occur when processors are forced to wait the others before to enter the communication phase of each super-step (in white). Graphs in the right are cumulative time (in percentage in ordinate) depicted for each processor point of view (abscissa) whereas graphs in the right are global points of view: cumulative times of each of the super-steps (time in ordinate). We also show the percentage (ordinate) of main memory used by the program (average of the processors) during the execution time of the program (abscissa).

Figure 4.14 shows the execution times for two scenarios for each protocol; the depicted results are fair witnesses of what we could observe from the large number of scenarios we have actually run. In the figure, the total execution time is split into three parts: the computation time (black) that essentially corresponds to the computation of successor states on each processor; the global and thus collective communication time (gray) that corresponds to states exchange; the waiting times (white) that occur when processors are forced to wait the others before to enter the communication phase of each super-step. Notice that because of the BSP model, these costs are obtained by considering the maximum times among the processors within each super-step, accumulated over the whole computation.

We can see on these graphs that the overall performance of our last algorithm (right-most bars) is always very good compared to the naive algorithm (left-most bars). In particular, the communication and waiting times are always greatly reduced. This holds for large state spaces as well as for smaller ones.

An important waiting time corresponds to an unbalanced computation: if some processors spend more time computing successors, the others will have to wait for them to finish this

computation before every processor enters the communication phase. In several occurrences, we can observe that, by increasing the local computation, we have worsen the balance, which increased the waiting time. This corresponds to graphs where the middle part in the second column is taller than the same part in the left column. However, we can observe that our last optimisation to improve the balance, without introduce an overhead of communications, is always very efficient and results in negligible waiting time in every case. The variations of observed computation times are similarly caused by a bad balance because we depicted the accumulation of the maximum times among the processors.

Finally, by comparing the left and right columns of results, we can observe that the overall speedup is generally better when larger state spaces are computed. This is mainly due to the fact that the waiting time accumulation becomes more important on longer runs.

We can see on these graphs that for "balance" the communications are always greatly reduced but some time a greater waiting times: this is due to the computation of the histograms and to the fact that we perform an heuristic (of the bin packing problem) for dispatching the classes of states on the processors and some classes contains states that induce a little bigger number of successors (and the probability that these states are regrouped on the same classes is greater in "balance" than in the complete random distribution of "naive"). Note that the hashing (completely random) of "naive" gives the better balancing on some scenarios. For a small OR scenario, the waiting time of "naive" is greater but more balanced. However, for a bigger scenario, "balance" outperforms "naive".

By measuring the memory consumption of our implementations, we could confirm the benefits of "balance" (emptied memory regularly) when large state spaces are computed. For instance, in the NS-2-2 scenario, we observed an improvement of the peak memory usage from 50% to 20% (maximum among all the processors). Similarly, for the WLP-1-2-1_2, the peak decreases so that the computation does not swap. For Y-3-2-1, "balance" used a little less memory but that enough to not crash the whole machine.

Notice that the memory use never decrease even for "balance". This is due to the GC strategy of Python for sets which de-allocate pages of the main memory only when no enough memory is available: allocated pages are directly used for other new items.

As a last observation about our "balance" implementation, we would like to emphasise that we observed a linear speedup with respect to the number of processors. In general, most parallel algorithms suffer from an amortised speedup (that happens for the "naive" implementation) when the number of processors increases. This is almost always caused by the increasing amount of communication that becomes dominant over the computation. Because our balance implementation is specifically dedicated to reduce the number of cross transitions, and thus the amount of communication, this problem is largely alleviated and we could observe amortised speedup only for very small models for which the degree of intrinsic parallelism is very reduced but whose state space is in any way computed very quickly.

4.4 LTL and CTL*'s benchmarks

In order to evaluate our algorithm, we have used two formulas of the form $\varphi U \mathsf{deadlock}$, where $\mathsf{deadlock}$ is an atomic proposition that holds iff state has no successor and φ is a formula that checks for an attack on the considered protocol: Fml1 is the classical "secrecy" and Fml2 is "aliveness" [62]. The chosen formulas globally hold so that the whole proof graph is computed. Indeed, on several instances with counterexamples, we have observed that the sequential algorithm can be faster than the parallel version when a violating state can be found quickly: our parallel algorithm uses a global breadth-first search while the sequential exploration is depth-first, which usually succeeds earlier. But when all the exploration has to be performed, which is widely acknowledged as the hardest case, our algorithm is always much faster. Moreover, we sometimes could not compute the state space sequentially while the distributed version suc-

ceeded, thanks to the distribution of states and sweep-line strategy — which is also used for sequential computing.

We have implemented a prototype version in Python, using SNAKES [178] for the Petri net part (which also allowed for a quick modelling of the protocols, including the Dolev-Yao attacker) and a Python BSP library [128] for the BSP routines (which are close to an MPI "alltoall"). We actually used the MPI version (with MPICH) of the BSP-Python library. While largely suboptimal (Python programs are interpreted and there is no optimisation about the representation of the states in SNAKES and the implementation of the attacker is not optimal at all), this prototype nevertheless allows an accurate *comparison* for acceleration. The benchmarks presented below have been performed using a cluster with 20 PCs connected through a 1 Gigabyte Ethernet network. Each PC is equipped with a 2GHz Intel® Pentium® dual core CPU, with 2GB of physical memory. This allowed to simulate a BSP computer with 40 processors equipped with 1GB of memory each.

Our case studies involved the following four protocols: (1) Needham-Schroeder public key protocol for mutual authentication; (2) Yahalom key distribution and mutual authentication using a trusted third party; (3) Otway-Rees key sharing using a trusted third party; (4) Kao-Chow key distribution and authentication. These protocols and their security issues are documented at the Security Protocols Open Repository (SPORE[4]).

As a last observation about our algorithm, we would like to emphasise that we observed a relative speedup with respect to the number of processors. In general, most parallel algorithms suffer from an amortised speedup when the number of processors increases. This is almost always caused by the increasing amount of communication that becomes dominant over the computation. Because our algorithm is specifically dedicated to reduce the number of cross transitions, and thus the amount of communication, this problem is largely alleviated and we could observe amortised speedup only for very small models for which the degree of intrinsic parallelism is very reduced but whose state space is in any way computed very quickly. Finally, measuring the memory consumption of our various algorithms, we could also confirm the benefits of our sweep-line implementation when large state spaces are computed.

Figure 4.19 gives the speed-up for each the two formulas and two sessions of each protocol. For the Yahalom protocol, the computation fails due to a lack of main memory (swapping) if less that 4 nodes are used: we could thus not give the speedup but only times. We observed a relative speedup with respect to the number of processors. Finally, measuring the memory consumption of our algorithm, we could also confirm the benefits of our sweep-line implementation when large state spaces are computed.

Figure 4.20 gives the timings for formula that checks for a typical attack of the protocols and for sessions with two honest agents.

[4] http://www.lsv.ens-cachan.fr/Software/spore

```
net Alice (this, agents, server, session) :
    buffer peer : int = ()
    buffer peer_nonce : Nonce = ()
    buffer keyAB : object = ()
    [agents?(B), peer+(B), snd+(this, server, this, B, Nonce((this, session)))] #1->
    ; [peer?(B),
       rcv?(B, this,
            ("crypt", ("secret", server, this), this, B, Na, key),
            ("crypt", key, Na), Nb),
       peer_nonce+(Nb), keyAB+(key) if Na == Nonce((this, session))] #3<-
    ; [peer?(B), peer_nonce?(Nb), keyAB?(key),
       snd+(this, B, ("crypt", key, Nb))] #4->

net Bob (this, server, session) :
    buffer peer : int = ()
    buffer peer_nonce : Nonce = ()
    buffer illisible : object = ()
    buffer keyAB : object = ()
    [rcv?(server, this, myster,
          ("crypt", ("secret", server, this), A, B, Na, key)),
     peer+(A), peer_nonce+(Na), illisible+(myster), keyAB+(key)] #2<-
    ; [peer?(A), peer_nonce?(Na), illisible?(myster), keyAB?(key),
       snd+(this, A, myster, ("crypt", key, Na), Nonce((this, session)))] #3->
    ; [peer?(A), keyAB?(key), rcv?(A, this, ("crypt", key, Nb)) if Nb == Nonce((this, session))]

net Server (this) :
    buffer peer_alice : int = ()
    buffer peer_bob : int = ()
    buffer peer_alice_nonce : Nonce = ()
    [rcv?(A, this, A, B, Na), peer_alice+(A), peer_alice_nonce+(Na), peer_bob+(B)] #1<-
    ; [peer_alice?(A), peer_alice_nonce?(Na), peer_bob?(B),
       snd+(this, B,
            ("crypt", ("secret", this, A), A, B, Na, ("secret", A, B, Na)),
            ("crypt", ("secret", this, B), A, B, Na, ("secret", A, B, Na)))] #2->

net Mallory (this, set_sessions) :
    buffer spy : object = Spy(
        (int, int, int, int, Nonce), #1
        (int, int,
         ("crypt", ("secret", int, int),
          int, int, Nonce, ("secret", int, int, Nonce)),
         ("crypt", ("secret", int, int),
          int, int, Nonce, ("secret", int, int, Nonce))), #2
        (int, int,
         ("crypt", ("secret", int, int),
          int, int, Nonce, ("secret", int, int, Nonce)),
         ("crypt", ("secret", int, int, Nonce), Nonce),
         Nonce), #3
        (int, int,
         ("crypt", ("secret", int, int, Nonce), Nonce)) #4
        )
    [rcv≪ ((this,)
           + tuple(range(1, this))
           + tuple(Nonce((this, s)) for s in set_sessions)
           )]
    ; ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.6. Kao Chow protocol in ABCD.

```
net Alice (A, agents, S, session) :
    buffer B_ : int = ()
    # M = Nonce((A, session))
    [agents?(B), B_+(B),
        snd+(Nonce((A, session)), A, B, ("crypt", ("secret", A, S), Nonce(A), Nonce((A, session)), A, B))] # 1->
    ; [rcv?(M, ("crypt", ("secret", A, S), Na, key))
       if M == Nonce((A, session)) and Na == Nonce(A)] # 4<-

net Bob (B, S) :
    buffer A_ : int = ()
    buffer M_ : Nonce = ()
    buffer myster_ : object = ()
    buffer kab_ : tuple = ()
    [rcv?(M, A, B, myster), A_+(A), M_+(M), myster_+(myster)] # 1<-
    ; [A_?(A), M_?(M), myster_?(myster),
        snd+(M, A, B, myster, ("crypt", ("secret", B, S), Nonce(B), M, A, B))] # 2->
    ; [M_?(M), rcv?(M, myster, ("crypt", ("secret", B, S), Nb, kab)),
                myster_+(myster), kab_+(kab) if Nb == Nonce(B)] # 3<-
    ; [A_?(A), M_?(M), myster_?(myster),
                snd+(M, myster)] # 4->

net Server (S) :
    buffer A_ : int = ()
    buffer B_ : int = ()
    buffer Na_ : Nonce = ()
    buffer Nb_ : Nonce = ()
    buffer M_ : Nonce = ()
    [rcv?(M, A, B,
          ("crypt", ("secret", A, S), Na, M, A, B),
          ("crypt", ("secret", B, S), Nb, M, A, B)),
     A_+(A), B_+(B), Na_+(Na), Nb_+(Nb), M_+(M)] # 2<-
    ; [A_?(A), B_?(B), Na_?(Na), Nb_?(Nb), M_?(M),
       snd+(M,
            ("crypt", ("secret", A, S), Na, ("secret", Na, Nb)),
            ("crypt", ("secret", B, S), Nb, ("secret", Na, Nb)))] # 3->, Kab=("secret", Na, Nb)


net Mallory (this, set_sessions) :
    buffer spy : object = Spy(
                                (Nonce, int, int, ("crypt", ("secret", int, int), Nonce, Nonce, int, int)), #1
                                (Nonce, int, int,
                                        ("crypt", ("secret", int, int), Nonce, Nonce, int, int),
                                        ("crypt", ("secret", int, int), Nonce, Nonce, int, int)), #2
                                (Nonce,
                                        ("crypt", ("secret", int, int), Nonce, tuple),
                                        ("crypt", ("secret", int, int), Nonce, tuple)), #3
                                (Nonce, ("crypt", ("secret", int, int), Nonce, tuple)) #4
                               )
    [rcv≪ ((this, Nonce(this))
           + tuple(range(1, this))
           + set_sessions
           )]
    ; ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.7. Otway Rees protocol in ABCD.

```
net Alice (A, agents, S) :
    buffer B_ : int = ()
    buffer Nb_ : Nonce = ()
    buffer keyAB_ : tuple = ()
    buffer myster_ : object = ()
    [agents?(B), B_+(B), snd+(A, Nonce(A))] # 1->
    ; [B_?(B), rcv?(("crypt", ("secret", A, S), B, keyAB, Na, Nb), myster),
       Nb_+(Nb), keyAB_+(keyAB), myster_+(myster) if Na == Nonce(A)] # 3<-
    ; [B_?(B), myster_?(myster), Nb_?(Nb), keyAB_?(keyAB),
       snd+(myster, ("crypt", keyAB, Nb))] # 4->

net Bob (B, S) :
    buffer A_ : int = ()
    buffer Na_ : Nonce = ()
    [rcv?(A, Na), A_+(A), Na_+(Na)] #1 <-
    ; [A_?(A), Na_?(Na), snd+(B, ("crypt", ("secret", B, S), A, Na, Nonce(B)))] #2 ->
    ; [A_?(A), rcv?(("crypt", ("secret", B, S), A, keyAB),
                    ("crypt", keyAB, Nb)) if Nb == Nonce(B)] # 4<-

net Server (S) :
    buffer A_ : int = ()
    buffer B_ : int = ()
    buffer Na_ : Nonce = ()
    buffer Nb_ : Nonce = ()
    [rcv?(B, ("crypt", ("secret", B, S), A, Na, Nb)), A_+(A),
     B_+(B), Na_+(Na), Nb_+(Nb)] # 2 <-
    ; [A_?(A), B_?(B), Na_?(Na), Nb_?(B),
       snd+(("crypt", ("secret", A, S), B, ("secret", Na, Nb), Na, Nb),
            ("crypt", ("secret", B, S), A, ("secret", Na, Nb)))] # 3-> # kab = (Na, Nb)

net Mallory (this) :
    buffer spy : object = Spy(
                                (int, Nonce), #1
                                (int, ("crypt", ("secret", int, int), int, Nonce, Nonce)), #2
                                (("crypt", ("secret", int, int), int, ("secret", Nonce, Nonce), Nonce, Nonce),
                                  ("crypt", ("secret", int, int), int, ("secret", Nonce, Nonce))), #3
                                (("crypt", ("secret", int, int), int, ("secret", Nonce, Nonce)),
                                  ("crypt", ("secret", Nonce, Nonce), Nonce)) #4
                              )
    [rcv≪ ((this, Nonce(this))
           + tuple(range(1, this))
           )]
    ; ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.8. Yahalom protocol in ABCD.

```
net Alice (A, agents, S) :
    buffer B_ : int = ()
    buffer Nb_ : Nonce = ()
    [agents?(B), B_+(B), snd+(A)] # 1−>
    ⨟ [rcv?(Nb), Nb_+(Nb)] # 2<−
    ⨟ [Nb_?(Nb), snd+(("crypt", ("secret", A, S), Nb))] # 3−>

net Bob (B, S) :
    buffer A_ : int = ()
    buffer myster_ : object = ()
    [rcv?(A), A_+(A)] # 1<−
    ⨟ [snd+(Nonce(B))] # 2−>
    ⨟ [rcv?(myster), myster_+(myster)] #3<−
    ⨟ [A_?(A), myster_?(myster), snd+(("crypt", ("secret", B, S), A, myster))] # 4−>
    ⨟ [rcv?(("crypt", ("secret", S, B), Nb)) if Nb == Nonce(B)] # 5<−

net Server (S) :
    buffer B_ : int = ()
    buffer Nb_ : Nonce = ()
    [rcv?(("crypt", ("secret", B, S), A, ("crypt", ("secret", A, S), Nb))), B_+(B), Nb_+(Nb)] #4<−
    ⨟ [B_?(B), Nb_?(Nb), snd+(("crypt", ("secret", S, B), Nb))] #5−>

net Mallory (this) :
    buffer spy : object = Spy(
                              (int),
                              (Nonce),
                              (("crypt", ("secret", int, int), Nonce)),
                              (("crypt", ("secret", int, int), int, ("crypt", ("secret", int, int), Nonce))),
                              (("crypt", ("secret", int, int), Nonce))
                              )
    [rcv≪ ((this, Nonce(this))
          + tuple(range(1, this))
          )]
    ⨟ ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.9. Woo and Lam protocol in ABCD.

```
buffer snd : object = ()
buffer rcv : object = ()

buffer ttA : int = 0
buffer ttS : int = 0
buffer ttB : int = 0

net Alice (A, agents, S, session) :
    buffer B_ : int = ()
    [agents?(B), B_+(B), ttA−(Ta),
     snd+(A, server, A, ("crypt", ("secret", A, S), Ta+1, B, ("secret", A, B, session))),
     ttA+(Ta+1)] #1−>


net Bob (B, S) :
    [ttB−(Tb), rcv?(S, B, ("crypt", ("secret", S, B), Ts, A, key)), ttB+(Ts)
     if Tb < Ts] # 2<−

net Server (S) :
    buffer A_ : int = ()
    buffer B_ : int = ()
    buffer keyAB : tuple = ()
    [ttS−(Ts), rcv?(A, S, A, ("crypt", ("secret", A, S), Ta, B, keyAB)),
     A_+(A), B_+(B), keyAB+(key), ttS+(Ta) if Ts < Ta] #<−1
    ⨾ [ttS−(Ts), A_?(A), B_?(B), keyAB?(key),
       snd+(S, B, ("crypt", ("secret", S, B), Ts+1, A, key)), ttS+(Ts+1)] # 2−>

net Mallory (this, set_sessions) :
    buffer spy : object = Spy(
                                (int, int, int, ("crypt", ("secret", int, int), int, int, ("secret", int, int, int))), #1
                                (int, int, ("crypt", ("secret", int, int), int, int, ("secret", int, int, int))) #2
                               )
    [rcv≪ ((this)
           + tuple(range(1, this))
           + tuple(range(0,3⊛ max(set_sessions))) # ensemble de time_stamp
           )]
    ⨾ ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.10. Wide Mouthed Frog in ABCD.

```
buffer snd : object = ()
buffer rcv : object = ()

net Alice (A, agents) :
    buffer _B : int = ()
    buffer _Nb : Nonce = ()
    [agents?(B), _B+(B), snd+(A, B, A, ("crypt", ("secret", A, B), Nonce(A)))] #1->
    ; [_B?(B), rcv?(B, A, ("crypt", ("secret", A, B), ("succ", Na), Nb)), _Nb+(Nb) if Na == Nonce(A)] #<-2
    ; [_B?(B), _Nb?(Nb), snd+(A, B, ("crypt", ("secret", A, B), ("succ", Nb)))] #->3
    ; [_B?(B), _Nb?(Nb), rcv?(B, A, ("crypt", ("secret", A, B), new_key, Nb_2))] #<-4

net Bob (B) :
    buffer _A : int = ()
    buffer _Na : Nonce = ()
    [rcv?(A, B, A, ("crypt", ("secret", A, B), Na)), _A+(A), _Na+(Na)] #1<-
    ; [_A?(A), _Na?(Na), snd+(B, A, ("crypt", ("secret", A, B), ("succ", Na), Nonce(B)))] #->2
    ; [_A?(B), rcv?(A, B, ("crypt", ("secret", A, B), ("succ", Nb))) if Nb == Nonce(B)] #<-3
    ; [_A?(A), _Na?(Na), snd+(B, A, ("crypt", ("secret", A, B), ("secret", Na, Nonce(B)), Nonce(A+B)))] #->4

net Mallory (this, init) :
    buffer spy : object = Spy(
                                (int, int, int, ("crypt", ("secret", int, int), Nonce)), #1->
                                (int, int, ("crypt", ("secret", int, int), ("succ", Nonce), Nonce)), #->2
                                (int, int, ("crypt", ("secret", int, int), ("succ", Nonce))), #->3
                                (int, int, ("crypt", ("secret", int, int), ("secret", Nonce, Nonce), Nonce)) #->4
                            )
    [rcv≪ ((this, Nonce(this))
          + tuple(range(1, this))
          + init)]
    ; ([spy?(s), snd−(m), rcv≫ (k), rcv≪ (s.learn(m, k))] ⊛ [False])
```

Figure 4.11. Andrew Secure RPC in ABCD.

```
alice.[rcv?("crypt", ("pub", this), Na, Nb), peer_nonce+(Nb) if Na == Nonce(this)]
bob1.[rcv?("crypt", ("pub", this), A, Na), peer+(A), peer_nonce+(Na)]
bob1.[rcv?("crypt", ("pub", this), Nb) if Nb == Nonce(this)]
bob2.[rcv?("crypt", ("pub", this), A, Na), peer+(A), peer_nonce+(Na)]
bob2.[rcv?("crypt", ("pub", this), Nb) if Nb == Nonce(this)]
```

Figure 4.12. File of the transition of reception of the Classical Needham Schroeder protocol in ABCD.

```
alice.peer
alice.peer_nonce
bob1.peer
bob1.peer_nonce
bob2.peer
bob2.peer_nonce
```

Figure 4.13. File of the designated places of the Classical Needham Schroeder protocol in ABCD.

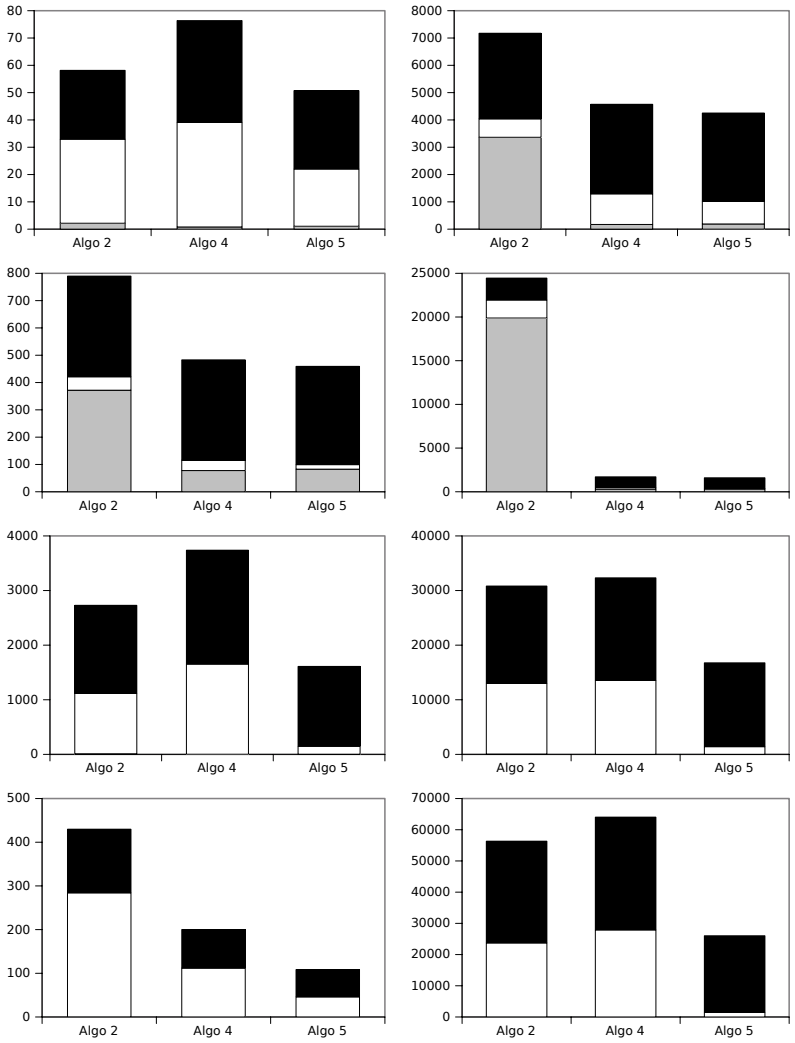


Figure 4.14. Computation times (in seconds) of Algorithms 2.2, 2.6 and 2.8 for the four studied protocols. Top row: two instances of NS yielding respectively about 8K (left) and 5M states (right). Second row: two instances of Y with about 400K (left) and 1M states (right). Third row: two instances of OR with about 12K (left) and 22K states (right). Bottom row: two instances of KC with about 400 (left) and 2K states (right).

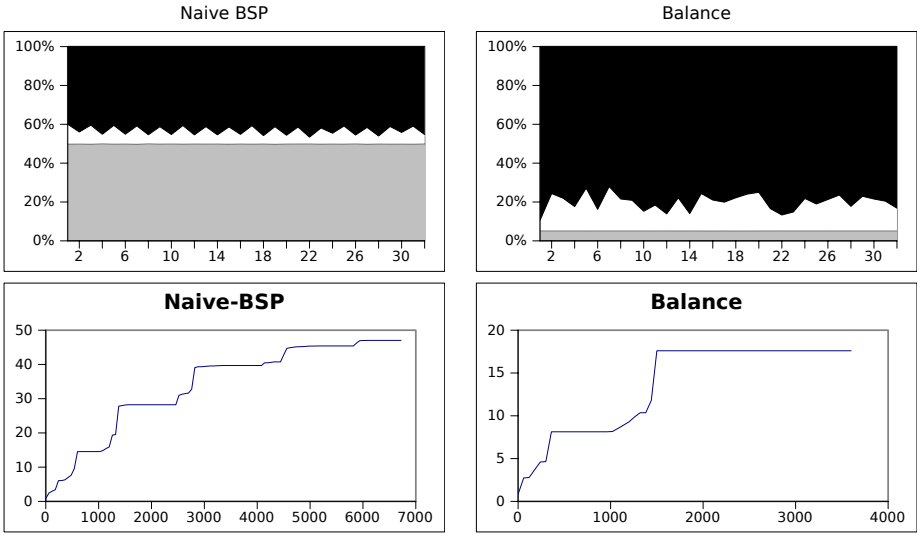


Figure 4.15. Performances for NS-2-2.

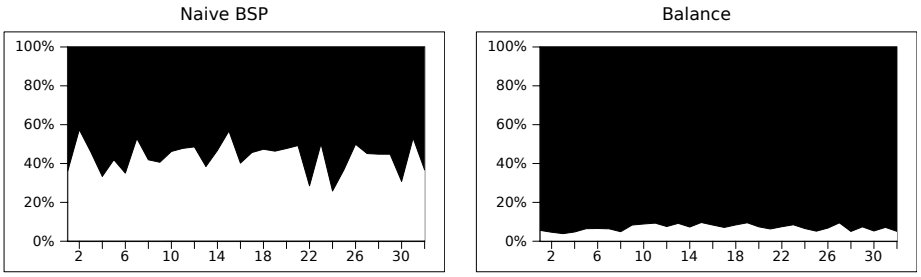


Figure 4.16. Performances for OR-1-2-1_2.

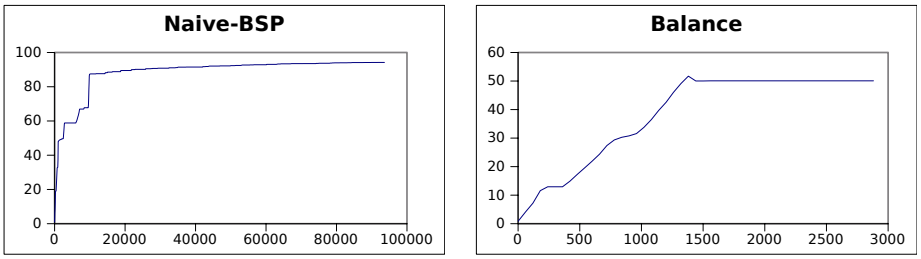


Figure 4.17. Performances for WLP-1-2-1_2.

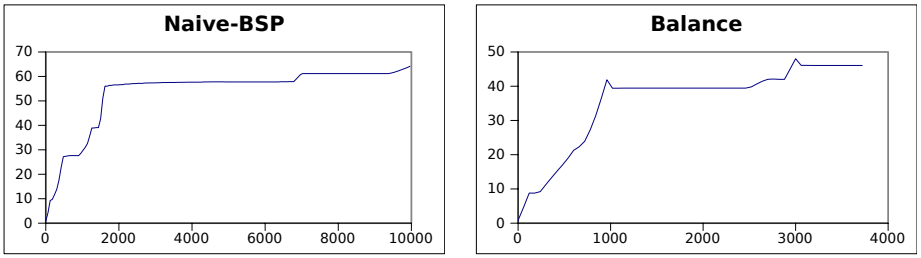


Figure 4.18. Performances for Y-3-2-1.

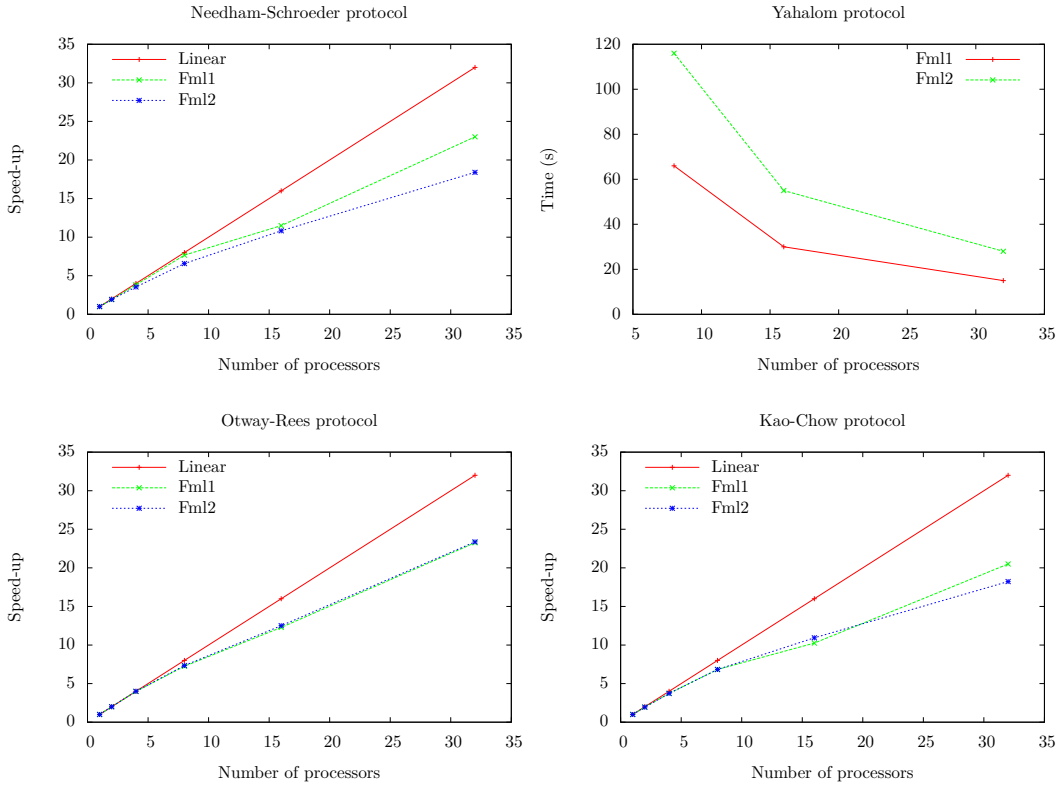


Figure 4.19. Timings depending on the number of processors for four of the protocols studied and where Fml1 is "secrecy" and Fml2 "aliveness".

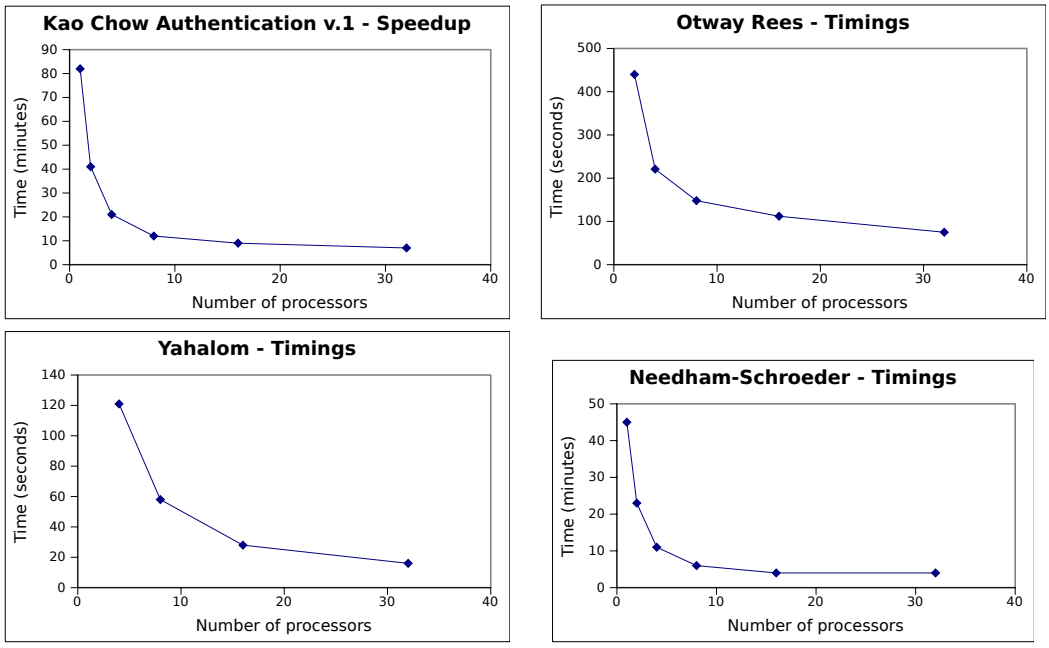


Figure 4.20. Timings depending on the number of processors for four of the protocols studied.

5 Conclusion

Designing security protocols is complex and often error prone: various attacks are reported in the literature to protocols thought to be "correct" for many years. This is due to the nature of protocols: they are executed as multiple concurrent sessions in an uncertain environment, where all messages flowing the network could be manipulating by an attacker which does not need to break cryptography. Indeed, protocols are broken merely because of attackers exploiting flaws in the protocols.

Each security protocol is designed to achieve certain goals after the execution. Those goals are called security properties. There are various security properties, for example, to ensure that secret data is not revealed to irrelevant parties. Due to the presence of an attacker, some protocols can not be able to preserve the expected security properties. Therefore it is very important to find a formal way to find flaws and to prove their correctness with respect to security properties.

To check if a protocol or a session of a protocol does not contain flaw, we have proposed to resort to model-checking, using an algebra of coloured Petri nets called ABCD to model the protocol, together with security properties that could be expressed as reachability properties, LTL, or CTL* formulas. Reachability properties lead to construct the state space of the model (*i.e.* the set of its reachable states). LTL and CTL* involve the construction of the state graph (*i.e.* the reachable states together with the transitions from one state to another) that is combined with the formula under analysis into a so called proof graph. In both cases, on-the-fly analysis allows to stop states explorations as soon as a conclusion can be drawn.

However, in general, this leads to compute a number of states that may be exponentially larger than the size of the model, which is the so called state space explosion problem. The critical problem of state space or state graph construction is to determine whether a newly generated state has been explored before. In a serial implementation this question is answered by organizing known states in a specific data-structure, and looking for the new states in that structure. As this is a centralized activity, a parallel or distributed solution must find an alternative approach. The common method is to assign states to processors using a static partition function which is generally a hashing of the states [102]. After a state has been generated, it is sent to its assigned location, where a local search determines whether the state already exists. Applying this method to security protocols fails in two points. First the number of cross-transitions (*i.e.* transitions between two states assigned to distinct processors) is too high and leads to a too heavy network use. Second, memorizing all of them in the main memory is impossible without crashing the whole parallel machine and is not clear when it is possible to put some states in disk and if heuristics [86, 148] would work well for complex protocols.

Our first solution is to use the well-structured nature of the protocols to choose which part of the state space is really needed for the partition function and to empty the data-structure in each super-step of the parallel computation. Our second solution entails automated classification of states into classes, and dynamic mapping of classes to processors. We find that both our methods execute significantly faster and achieve better network use than classical method. Furthermore, we find that the method that balances states does indeed achieve better network use, memory balance and runs faster.

The fundamental message is that for parallel discrete state space generation, exploiting certain

characteristics of the system and structuring the computation is essential. We have demonstrated techniques that proved the feasibility of this approach and demonstrated its potential. Key elements to our success were (1) an automated classification that reduces cross-transitions and memory use and growth locallity of the computations (2) using global barriers (which is a low-overhead method) to compute a global remappings and thus balancing workload and achieved a good scalability for the discrete state space generation of security protocols.

Then, we have shown how these ideas about state space computation could be generalized to the computation and analysis of proof graphs. The structure of state space exploration is preserved but enriched with the construction of the proof graph and its on-the-fly analysis. In the case of LTL, we could show that the required information to conclude about a formula is either available locally to a processor (even when states are dumped from the main memory at each super step), or is not needed anymore when a cross-transition occurs. Indeed, we have seen that no cross-transition occurs within a strongly connected component, which are the crucial structures in proof graphs to conclude about formulas truths. In the case of CTL* however, local conclusions can need to be delayed until a further recursive exploration is completed, which may occur on another processor. Rather than continuing such an exploration on the same processor, which would limit parallelism, we could design a way to organize the computation so that inconclusive nodes in the proof graph can be kept available until a conclusion comes from a recursive exploration, allowing to dump them immediately from the main memory. This more complex bookkeeping appears necessary due to the recursive nature of CTL* checking that can be regarded as nested LTL analysis.

5.1 Summary of contributions

Throughout this thesis, we have proposed several contributions summarized thereafter.

Models of several classical security protocols. Using the ABCD algebra, have been provided, showing quite a systematic style of modeling. In particular, the same model of a Dolev-Yao attacker can be reused in every cases. But more generally, modeling new protocols looks quite straightforward because they are very likely to reuse the same patterns as in the protocols we have modeled.

A parallel algorithm for state space generation. We have featuring an automated classification of states on processors, dynamic re-balancing of workload, sweep-line method to discharge unneeded states from the processors' memory. Experiments have also shown that this algorithm has limited network usage as well as a good scalability.

A parallel algorithm for LTL analysis. Bsed on the algorithm for state space exploration and inherits is good characteristics.

A generalization to CTL* of the parallel LTL analysis. CTL* has been studied also. With respect to the previous algorithms, this one uses a more complex memory management and requires to keep more states in memory, due to the nature of CTL* model-checking.

Prototype implementations of our algorithms. Implementations have been made and used to experiment on the modeled protocols. We have used the Python programming language for this purpose, which, being an interpreted language, does not allow to assess efficiency but, however, is perfectly suitable for evaluating the parallel performances of our algorithms.

A systematic performance analysis of our algorithms. Benchmarks have been conduced using various instances of the modeled protocols. This allowed to confirm their good parallel behavior, in particular scalability with respect to the number of processors.

5.2 Future works

Future works will be dedicated to build a real and efficient implementation from our prototypes. It will feature in particular a CTL* model-checker, allowing to verify non-trivial security properties. Using this implementation, we would like to run benchmarks in order to compare our approach with existing tools. We would like also to test our algorithm on parallel computer with more processors in order to confirm the scalability that we could observe on 40 processors.

Another way to improve performances will be to consider symbolic state space representations as well as symbolic state space computation. In the former case, we are targeting in particular representations based on decision diagrams. In the latter case, we are thinking about adapting symmetry or partial order reduction methods to reduce the number of executions that need to be explored. Reductions methods appear to be the simplest step because they somehow result in exploring less states. Yet, they usually result in an exponential reduction of the number of computed states or transitions. On the other hand, using symbolic representations looks more challenging because storing large number of states in such structures is computationally efficient only when we can also apply a symbolic successor function, *i.e.* compute the successors of sets of states instead of those of a single state.

Moreover, we are working on the formal proofs of our algorithms. Proving a verification algorithm is highly desirable in order to certify the truth of the diagnostics delivered by such an algorithm. Such a proof is possible because, thanks to the BSP model, our algorithm remains simple in its structure.

Finally, we would like to generalise our present results by extending the application domain. In the security domain, we will consider more complex protocols with branching and looping structures, as well as complex data types manipulations. In particular, we will consider protocols for secure storage distributed through peer-to-peer communication [184].

Bibliography

[1] Xavier Allamigeon and Bruno Blanchet. Reconstruction of attacks against cryptographic protocols. In *Computer Security Foundations Workshop (CSFW)*, pages 140–154. IEEE Computer Society, 2005. Page 29.

[2] S. Allmaier, S. Dalibor, and D. Kreische. Parallel graph generation algorithms for shared and distributed memory machines. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Proceedings of Parallel Computing (ParCo)*, volume 12, pages 581–588. Elsevier, 1997. Page 38.

[3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002. Page 61.

[4] R. M. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In C. Palamidessi, editor, *Concur*, volume 1877 of *LNCS*, pages 380–394. Springer-Verla, 2000. Page 4.

[5] Christian Appold. Efficient symmetry reduction and the use of state symmetries for symbolic model checking. In Angelo Montanari, Margherita Napoli, and Mimmo Parente, editors, *Symposium on Games, Automata, Logic, and Formal Verification (GANDALF)*, volume 25 of *EPTCS*, pages 173–187, 2010. Page 34.

[6] Christian Appold. Improving bdd based symbolic model checking with isomorphism exploiting transition relations. In Giovanna D'Agostino and Salvatore La Torre, editors, *Symposium on Games, Automata, Logics and Formal Verification (GandALF)*, volume 54 of *EPTCS*, pages 17–30, 2011. Page 31.

[7] M. Arapinis, S. Delaune, and S. Kremer. From one session to many: Dynamic tags for security protocols. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 5330 of *LNCS*, pages 128–142. Springer, 2008. Page 39.

[8] A. Armando, R. Carbone, and L. Compagna. LTL model checking for security protocols. In *Proceedings of CSF*, pages 385–396. IEEE Computer Society, 2007. Page 28.

[9] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. *Applied Non-Classical Logics*, 19(4):403–429, 2009. Page 39.

[10] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Pages 2 and 28.

[11] A. Armando and *et* al. The AVISPA tool for the automated validation of Internet security protocols and applications. In K. Etessami and S. K. Rajamani, editors, *Proceedings of Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005. Pages 2, 28, 29 and 39.

[12] Alessandro Armando, Roberto Carbone, and Luca Compagna. Ltl model checking for security protocols. *Journal of Applied Non-Classical Logics*, 19(4):403–429, 2009. Pages 2 and 27.

[13] Mathilde Arnaud. *Formal verification of secured routing protocols.* PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, 2011. Page 8.

[14] A.V.Gerbessiotis. *Topics in Parallel and Distributed Computation.* PhD thesis, Harvard University, 1993. Page 23.

[15] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003. Page 23.

[16] M. Bamha and G. Hains. Frequency-adaptive join for Shared Nothing machines. *Parallel and Distributed Computing Practices*, 2(3):333–345, 1999. Page 23.

[17] M. Bamha and G. Hains. An Efficient equi-semi-join Algorithm for Distributed Architectures. In V. Sunderam, D. van Albada, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2005)*, LNCS. Springer, 2005. Page 23.

[18] J. Barnat. *Distributed Memory LTL Model Checking.* PhD thesis, Faculty of Informatics Masaryk University Brno, 2004. Pages 37 and 113.

[19] J. Barnat, L. Brim, and I. Cëerná. Property driven distribution of nested dfs. In M. Leuschel and U. Ultes-Nitsche, editors, *Workshop on Verification and Computational Logic (VCL)*, volume DSSE-TR-2002-5, pages 1–10. Dept. of Electronics and Computer Science, University of Southampton (DSSE), UK, Technical Report, 2002. Page 36.

[20] D. Basin. How to evaluate the security of real-life cryptographic protocols? The cases of ISO/IEC 29128 and CRYPTREC. In *Workshop on Real-life Cryptographic Protocols and Standardization*, 2010. Pages 2 and 38.

[21] David A. Basin. Lazy infinite-state analysis of security protocols. In Rainer Baumgart, editor, *Secure Networking - CQRE (Secure), International Exhibition and Congress Düsseldorf*, volume 1740 of *LNCS*, pages 30–42. Springer, 1999. Page 29.

[22] David A. Basin, Sebastian Mödersheim, and Luca Viganò. Algebraic intruder deductions. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3835 of *LNCS*, pages 549–564. Springer, 2005. Page 29.

[23] David A. Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *Int. J. Inf. Sec.*, 4(3):181–208, 2005. Pages 29 and 39.

[24] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Programming distributed memory sytems using openmp. In *IPDPS*, pages 1–8, 2007. Page 19.

[25] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop*, volume 1066 of *LNCS*, pages 232–243. Springer, 1995. Page 28.

[26] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 142–155. Springer-Verlag, 1994. Page 30.

[27] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *omputer Security Foundations Symposium (CSF)*, pages 124–140. IEEE Computer Society, 2009. Page 8.

[28] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for ctl*. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 388–398. IEEE Computer Society, 1995. Pages 30, 39, 57, 61, 63, 64, 65, 67, 68, 73 and 116.

[29] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004. Pages 22, 23, 38 and 41.

[30] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994. Page 23.

[31] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE CSFW'01*. IEEE Computer Society, 2001. Pages 26 and 29.

[32] S. Blom, B. Lisser, J. van de Pol, and M. Weber. A database approach to distributed state space generation. *Electr. Notes Theor. Comput. Sci.*, 198(1):17–32, 2008. Page 38.

[33] S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In *ENTCS*, volume 89. Elsevier, 2003. Page 38.

[34] S. C. C. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. C. van de Pol. μ-CRL: A toolset for analysing algebraic specifications. In *Proceedings Computer Aided Verification (CAV)*, number 2102 in LNCS, pages 250–254, 2001. Page 28.

[35] Stefan Blom, Jan Friso Groote, Sjouke Mauw, and Alexander Serebrenik. Analysing the bke-security protocol with μcrl. *Electr. Notes Theor. Comput. Sci.*, 139(1):49–90, 2005. Page 28.

[36] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005. Page 29.

[37] R. Bouroulet, R. Devillers, H. Klaudel, E. Pelz, and F. Pommereau. Modeling and analysis of security protocols using role based specifications and Petri nets. In *ICATPN*, volume 5062 of *LNCS*, pages 72–91. Springer, 2008. Page 98.

[38] C. Boyd. Security architectures using formal methods. *IEEE journal on Selected Areas in Communications*, 11(5):684–701, 1993. Page 5.

[39] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999. Page 23.

[40] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical report 39, Digital Systems Research Center, 1989. Pages 103 and 105.

[41] Duncan Campbell. Further results with algorithmic skeletons for the clumps model of parallel computation, 1996. Page 25.

[42] Duncan K. G. Campbell. On the clumps model of parallel computation. *Inf. Process. Lett.*, 66(5):231–236, 1998. Page 25.

[43] Duncan K.G. Campbell. Clumps: A candidate model of efficient, general purpose parallel computation. Technical report, 1994. Page 25.

[44] F. Cappello, P. Fraigniaud, B. Mans, and A.L. Rosenberg. HiHCoHP toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. In *IEEE/ACM IPDPS'2001*. IEEE press, 2001. Page 26.

[45] Franck Cappello, Pierre Fraigniaud, Bernard Mans, and Arnold L. Rosenberg. An algorithmic model for heterogeneous hyper-clusters: rationale and experience. *Int. J. Found. Comput. Sci.*, 16(2):195–215, 2005. Page 26.

[46] A. Chan, F. Dehne, and R. Taylor. Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *Journal of High Performance Computing Applications*, 2005. Page 23.

[47] Barbara Chapman. *Using OpenMP: portable shared memory parallel programming.* The MIT Press, 2008. Page 19.

[48] Yannick Chevalier and Laurent Vigneron. Automated unbounded verification of security protocols. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification (CAV)*, LNCS, pages 324–337. Springer, 2002. Page 27.

[49] S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In T. Margaria and W. Yi, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001. Pages 46, 49, 53 and 55.

[50] M.-Y. Chung and G. Ciardo. A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation. In *ENTCS*, volume 135, pages 65–80. Elsevier, 2006. Page 38.

[51] J. Clark and J. Jacob. A survey of authentication protocol literature : Version 1.0. Available at `http://www-users.cs.york.ac.uk/~jac/papers/drareview.ps.gz`, 1997. Pages 4, 102 and 103.

[52] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In D.L. Dill, editor, *Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 415–427. Springer-Verlag, 1994. Pages 60 and 65.

[53] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *CAV'98*, volume 1427 of *LNCS*. Springer, 1998. Page 34.

[54] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. Page 32.

[55] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 2000. Pages 27 and 32.

[56] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996. Page 34.

[57] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Verifying security protocols with brutus. *ACM Trans. Softw. Eng. Methodol.*, pages 443–487, 2000. Pages 29 and 32.

[58] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Efficient verification of security protocols using partial-order reductions. *STTT*, pages 173–188, 2003. Page 32.

[59] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(121–147), 1993. Page 30.

[60] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. Page 20.

[61] H. Comon-Lundh and V. Cortier. How to prove security of communication protocols? a discussion on the soundness of formal models w.r.t. computational ones. In *STACS*, pages 29–44, 2011. Page 38.

[62] R. Corin. *Analysis Models for Security Protocols.* PhD thesis, University of Twente, 2006. Pages 39 and 119.

[63] Ricardo Corin, Sandro Etalle, Pieter H. Hartel, and Angelika Mader. Timed model checking of security protocols. In Vijayalakshmi Atluri, Michael Backes, David A. Basin, and Michael Waidner, editors, *Formal Methods in Security Engineering (FMSE)*, pages 23–32. ACM, 2004. Page 28.

[64] Ricardo Corin, Sandro Etalle, and Ari Saptawijaya. A logic for constraint-based security protocol analysis. In *Symposium on Security and Privacy*, pages 155–168. IEEE Computer Society, 2006. Page 29.

[65] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for verication of temporal properties. *Formal Methods in System design*, 1:275–288, 1992. Page 30.

[66] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for Petri net analysis. In *ICATPN'02*. Springer, 2002. Page 31.

[67] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 3731 of *LNCS*, pages 443–457. Springer, 2005. Page 31.

[68] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006. Pages 2, 28 and 29.

[69] Cas J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 414–418. Springer, 2008. Page 29.

[70] Cas J. F. Cremers and Sjouke Mauw. Checking secrecy by means of partial order reduction. In Daniel Amyot and Alan W. Williams, editors, *System Analysis and Modeling (SAM)*, volume 3319 of *LNCS*, pages 171–188. Springer, 2004. Page 29.

[71] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28:1–12, 1993. Page 24.

[72] M. Dam. Ctl and ectl as fragments of the modal mu-calculus. In *Colloquium on Trees and Algebra in Programming*, volume 581 of *LNCS*, pages 145–164. Springer-Verlag, 1992. Page 63.

[73] P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *Euro-Par'96. Parallel Processing*, 1996. Page 25.

[74] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999. Page 22.

[75] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996. Page 23.

[76] N. Deo and P. Micikevicius. Coarse-grained parallelization of distance-bound smoothing for the molecular conformation problem. In S. K. Das and S. Bhattacharya, editors, *4th International Workshop Distributed Computing, Mobile and Wireless Computing (IWDC)*, volume 2571 of *LNCS*, pages 55–66. Springer, 2002. Page 23.

[77] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. Pages 2, 7, 42 and 96.

[78] D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996. Page 23.

[79] N. Drosinos and N. Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In *Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–15, 2004. Page 21.

[80] R Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2), February 1990. Page 18.

[81] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*, 1999. Pages 26 and 39.

[82] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP), part of FLOC conference.*, 1999. Page 4.

[83] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing'01*, volume 2265 of *LNCS*. Springer, 2001. Page 112.

[84] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996. Page 34.

[85] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 2008. Page 34.

[86] S. Evangelista and L. M. Kristensen. Dynamic State Space Partitioning for External Memory Model Checking. In *Proceedings of Formal Methods In Computer Sciences (FMICS)*, volume 5825 of *LNCS*, pages 70–85. Springer, 2009. Pages 28 and 131.

[87] Sami Evangelista and Lars Michael Kristensen. Dynamic state space partitioning for external memory model checking. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5825 of *LNCS*, pages 70–85. Springer, 2009. Page 31.

[88] Sami Evangelista and Lars Michael Kristensen. Hybrid on-the-fly ltl model checking with the sweep-line method. In Serge Haddad and Lucia Pomello, editors, *Application and Theory of Petri Nets*, volume 7347 of *LNCS*, pages 248–267. Springer, 2012. Page 31.

[89] Sami Evangelista and Jean-François Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In Patrice Godefroid, editor, *Model Checking Software (SPIN)*, volume 3639 of *LNCS*, pages 43–57. Springer, 2005. Page 31.

[90] S. Even and O. Goldreich. On the security of multiparty ping pong protocols. In *24th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 1983. Page 4.

[91] Jonathan Ezekiel and Gerald Lüttgen. Measuring and evaluating parallel state-space exploration algorithms. *Electr. Notes Theor. Comput. Sci.*, 198(1):47–61, 2008. Pages 31 and 36.

[92] Jonathan Ezekiel, Gerald Lüttgen, and Gianfranco Ciardo. Parallelising symbolic state-space generators. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 268–280. Springer, 2007. Page 31.

[93] P Ferragina and F. Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24(3):177–194, 1999. Page 23.

[94] A. Ferreira, I. Guérin-Lassous, K. Marcus, and A. Rau-Chauplin. Parallel computation on interval graphs: algorithms and experiments. *Concurrency and Computation: Practice and Experience*, 14(11):885–910, 2002. Page 23.

[95] M.J. Flynn. Some computer organizations and their effectiveness. In *Trans. on Computers*, volume C-21(9), pages 948–960. IEEE, 1972. Page 18.

[96] Wan Fokkink, Mohammad Torabi Dashti, and Anton Wijs. Partial order reduction for branching security protocols. In Luís Gomes, Victor Khomenko, and João M. Fernandes, editors, *Conference on Application of Concurrency to System Design (ACSD)*, pages 191–200. IEEE Computer Society, 2010. Page 32.

[97] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM. Page 23.

[98] Matthew I. Frank, Anant Agarwal, and Mary K. Vernon. LoPC: modeling contention in parallel algorithms. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 276–287. ACM, 1997. Page 25.

[99] Mark A. Franklin. Vlsi performance comparison of banyan and crossbar communications networks. *IEEE Trans. Computers*, 30(4):283–291, 1981. Page 23.

[100] Peter Gammie and Ron van der Meyden. Mck: Model checking the logic of knowledge. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 479–483. Springer, 2004. Page 39.

[101] H. Gao. *Analysis of Security Protocols by Annotations*. PhD thesis, Technical University of Denmark, 2008. Pages 2 and 28.

[102] H. Garavel, R. Mateescu, and I. M. Smarandache. Parallel state space construction for model-checking. In M. B. Dwyer, editor, *Proceedings of SPIN*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001. Pages 35, 36, 37 and 131.

[103] I. Garnier and F. Gava. CPS Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons. *Parallel, Emergent and Distributed Systems*, 2011. To appear. Page 23.

[104] F. Gava, M. Guedj, and F. Pommereau. A BSP algorithm for the state space construction of security protocols. In *Ninth International Workshop on Parallel and Distributed Methods in Verification (PDMC 2010)*, pages 37–44. IEEE, 2010. Pages 41, 42 and 95.

[105] F. Gava, M. Guedj, and F. Pommereau. A bsp algorithm for on-the-fly checking ltl formulas on security protocols. In *Symposium on PArallel and Distributed Computing (ISPDC)*. IEEE, 2012. Page 57.

[106] F. Gava, M. Guedj, and F. Pommereau. Performance evaluations of a bsp algorithm for state space construction of security protocols. In *Euromicro Parallel and Distributed proceessing (PDP)*. IEEE, 2012. Page 95.

[107] T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Workshop on Issues in the Theory of Security (WITS)*, 2003. Page 28.

[108] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel priority queue and list contraction: The bsp approach. *Computing and Informatics*, 21:59–90, 2002. Page 23.

[109] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *International Symposium on Protocol Specification, Testing and Verification (IFIP)*, volume 38 of *IFIP Conference Proceedings*, 1995. Page 65.

[110] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel Research, 2007. Pages 21 and 23.

[111] P. B. Gibbons. A more practical pram model. In *SPAA*, 1989. Page 24.

[112] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation. In *SPAA'97 Symposium on Parallel Algorithms and Architectures*, pages 72–83, Newport, Rhode Island USA, June 1997. ACM. Page 25.

[113] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *Form. Methods Syst. Des.*, 7(3):227–241, 1995. Page 33.

[114] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004. Page 23.

[115] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press. Page 23.

[116] I. Gu'erin-Lassous and J. Gustedt. Portable List Ranking: an Experimental Study. *ACM Journal of Experiments Algorithms*, 7(7):1–18, 2002. Page 23.

[117] Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *J. Autom. Reasoning*, 48(2):159–195, 2012. Page 29.

[118] Alexandre Hamez. *Génération efficace de grands espaces d'états*. PhD thesis, University of Pierre and Marie Curry (LIP6), 2009. Page 38.

[119] Alexandre Hamez, Fabrice Kordon, and Yann Thierry-Mieg. libdmc: a library to operate efficient distributed model checking. In *Workshop on Performance Optimization for High-Level Languages and Libraries - associated to IPDPS'2007*, pages 495–504. IEEE Computer Society, 2007. Page 38.

[120] K. Hamidouche, A. Borghi, P. Esterie, J. Falcou, and S. Peyronnet. Three High Performance Architectures in the Parallel APMC Boat. In *Ninth International Workshop on Parallel and Distributed Methods in Verification (PDMC 2010)*, pages 20–27, 2010. Page 21.

[121] Olivier Heen, Gilles Guette, and Thomas Genet. On the unobservability of a trust relation in mobile ad hoc networks. In Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks (WISTP)*, volume 5746 of *LNCS*, pages 1–11. Springer, 2009. Page 8.

[122] Nevin Heintze, J. D. Tygar, Jeannette Wing, and H. Chi Wong. Model checking electronic commerce protocols. In *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, WOEC, pages 10–10. USENIX Association, 1996. Pages 30 and 39.

[123] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *SC*, 2000. Page 21.

[124] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proceedings of Computer Aided Verification (CAV)*, number 1855 in LNCS, pages 20–35, 2000. Page 37.

[125] Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation. i. the model. *J. Parallel Distrib. Comput.*, 16(3):212–232, 1992. Page 24.

[126] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998. Page 106.

[127] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998. Page 21.

[128] K. Hinsen. Parallel scripting with Python. *Computing in Science & Engineering*, 9(6), 2007. Pages 117 and 120.

[129] Gerard Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search (extended abstract). In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996. Page 64.

[130] G.J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(10):2413–2433, 1985. Page 33.

[131] G.J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C. West, editors, *Proc. 6th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, 1987. Page 33.

[132] J. Holzmann. *The Spin Model Checker*. Addison Wesley, 2004. Page 28.

[133] J. Hooman and J. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings of Symposium On Applied Computing (SAC)*, pages 351–258, 2002. Page 38.

[134] Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. In Bruce Hendrickson et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999. Page 23.

[135] C. P. Inggs. *Parallel Model Checking on Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Manchester, 2004. Pages 30 and 33.

[136] C. Norris Ip and David L. Dill. Verifying systems with replicated components in murφ. *Form. Methods Syst. Des.*, 14(3):273–310, 1999. Page 34.

[137] Sunu Antony Joseph. Evaluating threading building blocks pipelines, 2007. Page 20.

[138] B.H. H. Juurlink and H. A. G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Euro-Par, Parallel Processing*, 1996. Page 25.

[139] I. Lung Kao and R. Chow. An efficient and secure authentication protocol using uncertified keys. *Operating Systems Review*, 29(3):14–21, 1995. Page 102.

[140] Victor Khomenko and Maciej Koutny. Towards an efficient algorithm for unfolding petri nets. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Concurrency Theory (CONCUR)*, volume 2154 of *LNCS*, pages 366–380. Springer, 2001. Page 34.

[141] W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzinger. Probability, parallelism and the state space exploration problem. In R. Puigjaner, N. N. Savino, and B. Serra, editors, *Proceedings of Computer Performance Evaluation-Modeling, Techniques and Tools (TOOLS)*, number 1469 in LNCS, pages 165–179. Springer-Verlag, 1998. Page 37.

[142] Lars Michael Kristensen and Laure Petrucci. An approach to distributed state space exploration for coloured petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets (ICATPN)*, volume 3099 of *LNCS*, pages 474–483. Springer, 2004. Page 54.

[143] Peter Krusche and Alexander Tiskin. New algorithms for efficient parallel string comparison. In Friedhelm Meyer auf der Heide and Cynthia A. Phillips, editors, *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 209–216. ACM, 2010. Page 23.

[144] Alexey Kukanov. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4):309–322, 2007. Page 19.

[145] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In *ENTCS*, volume 128, pages 19–34. Elsevier, 2005. Page 37.

[146] I. Guerin Lassous. *Algorithmes paralleles de traitement de graphes: une approche basee sur l'analyse experimentale*. PhD thesis, University de Paris VII, 1999. Page 23.

[147] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006. Page 23.

[148] F. Lerda and R. Sista. Distributed-memory model checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of SPIN*, number 1680 in LNCS, pages 22–39. Springer-Verlag, 1999. Pages 37 and 131.

[149] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with Spin. In *SPIN'99*. Springer, 1999. Page 54.

[150] A. Lluch-Lafuente. implified distributed model checking by localizing cycles. Technical Report 176, Institute of Computer Science at Freiburg University, 2002. Page 38.

[151] Frédéric Loulergue, Frédéric Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In Vaidy S. Sunderam, Gaétan Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science (ICCS)*, LNCS 3515, pages 1046–1054. Springer, 2005. Page 106.

[152] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996. Page 28.

[153] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998. Pages 27 and 28.

[154] LSV, ENS Cachan. SPORE: Security protocols open repository. `http://www.lsv.ens-cachan.fr/Software/spore`. Page 43.

[155] Paolo Maggi and Riccardo Sisto. Using spin to verify security properties of cryptographic protocols. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking of Software (SPIN)*, volume 2318 of *LNCS*, pages 187–204. Springer, 2002. Page 28.

[156] Kiminori Matsuzaki. Efficient Implementation of Tree Accumulations on Distributed-Memroy Parallel Computers. In *Fourth International Workshop on Practical Aspects of High-Level Parallel Programming (PAPP 2007), part of The International Conference on Computational Science (ICCS 2007)*, 2007. to appear. Page 23.

[157] W. F. McColl. General purpose parallel computing. Oxford University Programming Research Group, April 1992. Page 25.

[158] Catherine Meadows. The nrl protocol analyzer: An overview. *J. Log. Program.*, 26(2):113–131, 1996. Pages 26 and 28.

[159] Catherine Meadows. Analysis of the internet key exchange protocol using the nrl protocol analyzer. In *IEEE Symposium on Security and Privacy*, pages 216–231, 1999. Page 28.

[160] Jonathan K. Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In Michael K. Reiter and Pierangela Samarati, editors, *Computer and Communications Security (CCS)*, pages 166–175. ACM, 2001. Page 29.

[161] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murphi. In *IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society, 1997. Page 27.

[162] Sebastian Mödersheim, Luca Viganò, and David A. Basin. Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *Journal of Computer Security*, 18(4):575–618, 2010. Pages 27, 29 and 98.

[163] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communication of the ACM*, 21(12), 1978. Page 99.

[164] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008. Page 20.

[165] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 4(2):153–167, 1997. Pages 28 and 37.

[166] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. Pages 26 and 28.

[167] S. Orzan, J. van de Pol, and M. Espada. A state space distributed policy based on abstract interpretation. In *ENTCS*, volume 128, pages 35–45. Elsevier, 2005. Page 37.

[168] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987. Page 102.

[169] Sam Owre and Natarajan Shankar. A brief overview of pvs. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*, pages 22–27. Springer, 2008. Page 26.

[170] C. Pajault. *Model Checking parallèle et réparti de réseaux de Petri colorés de haut-niveau*. PhD thesis, Conservatoire National des Arts et Métiers, 2008. Pages 37 and 38.

[171] Christophe Pajault and Jean-François Pradat-Peyre. Distributed colored petri net model-checking with cyclades. pages 347–361, 2006. Page 38.

[172] L. C. Paulson. Relations between secrets: Two formal analyses of the yahalom protocol. *J. Computer Security*, 2001. Page 104.

[173] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998. Page 28.

[174] D. Petcu. Parallel explicit state reachability analysis and state space construction. In *Proceedings of ISPDC*, pages 207–214. IEEE Computer Society, 2003. Page 37.

[175] J. L. Peterson. *Petri net theory*. Prentice Hall, 1981. Page 8.

[176] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften des Instituts fur instrumentelle Mathematik. Universitat Bonn, 1962. Page 8.

[177] Petri net markup language. `http://www.pnml.org`. Page 112.

[178] F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. In *Proc. of PNTAP'08*, ACM Digital Library, pages 1–10. ACM, 2008. Pages 39, 95, 110, 117 and 120.

[179] F. Pommereau. *Algebras of coloured Petri nets*. Habilitation thesis, University Paris-East Creteil, 2009. Pages 13, 14, 16 and 18.

[180] F. Pommereau. *Algebras of coloured Petri nets*. Habilitation thesis, University Paris-East CrÃlteil, 2009. Pages 42, 43 and 46.

[181] F. Pommereau. *Algebras of coloured Petri nets*. Lambert Academic Publisher, 2010. ISBN 978-3-8433-6113-2. Pages 8, 13, 42 and 95.

[182] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998. Page 23.

[183] M. Rusinowith and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In *14th Computer Security Foundations Workshop (CSFW)*, pages 174–190. IEEE, 2001. Page 39.

[184] S. Sanjabi and F. Pommereau. Modelling, verification, and formal analysis of security properties in a P2P system. In *Workshop on Collaboration and Security (COLSEC'10)*, IEEE Digital Library, pages 543–548. IEEE, 2010. Page 133.

[185] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1999. Page 7.

[186] H. Sivaraj and G. Gopalakrishnan. Random walk heuristic algorithms for distributed memory model checking. In *ENTCS*, volume 89. Elsevier, 2003. Page 37.

[187] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. Pages 22, 38 and 41.

[188] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998. Page 23.

[189] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI The Complete Reference*. MIT Press, 1996. Bibliothèque du LIFO. Page 20.

[190] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001. Pages 27 and 29.

[191] Xiaodong Dawn Song. *An automatic approach for building secure systems*. PhD thesis, University of California, Berkeley, 2002. Page 30.

[192] U. Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997. Page 28.

[193] U. Stern and D. L. Hill. Parallelizing the Murφ verifier. In O. Grumberg, editor, *Proceedings of Computer-Aided Verification (CAV)*, number 1254 in LNCS, pages 256–267. Springer-Verlag, 2997. Page 37.

[194] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, pages 146–160, 1972. Page 57.

[195] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1):191–230, 1999. Page 29.

[196] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998. Pages 23 and 109.

[197] M. Llanos Tobarra, Diego Cazorla, Fernando Cuartero, and Gregorio Díaz. Analysis of web services secure conversation with formal methods. In *International Conference on Internet and Web Applications and Services (ICIW)*, page 27. IEEE Computer Society, 2007. Page 8.

[198] M. Llanos Tobarra, Diego Cazorla, Fernando Cuartero, Gregorio Díaz, and María-Emilia Cambronero. Model checking wireless sensor network security protocols: Tinysec + leap + tinypk. *Telecommunication Systems*, 40(3-4):91–99, 2009. Page 8.

[199] Ashutosh Trivedi. Techniques in symbolic model checking, 2003. Page 36.

[200] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103, 1990. Page 26.

[201] B. Vergauwen and J. Lewi. A linear local model-checking algorithm for ctl. In E. Best, editor, *CONCUR*, volume 715 of *LNCS*, pages 447–461. Springer-Verlag, 1993. Pages 30 and 65.

[202] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008. Pages 26 and 28.

[203] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 1994. Page 104.

[204] A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming*. PhD thesis, Universita degli studi di Pisa, 1999. Page 18.

Softwares

1 http://openmp.org
2 http://threadingbuildingblocks.org
3 http://developer.nvidia.com/category/zone/cuda-zone
4 http://maude.cs.uiuc.edu/tools/Maude-NPA/
5 http://www.cl.cam.ac.uk/research/hvg/isabelle/
6 http://pvs.csl.sri.com
7 http://www.cs.ox.ac.uk/people/gavin.lowe/Security/Casper
8 http://www.cs.ox.ac.uk/projects/concurrency-tools
9 http://homepages.cwi.nl/ mcrl/
10 http://www.uppaal.org/
11 http://spinroot.com/spin/whatispin.html
12 http://www.proverif.ens.fr
13 http://www2.imm.dtu.dk/cs_LySa/lysatool
14 http://www.avispa-project.org
15 http://people.inf.ethz.ch/cremersc/scyther/
16 `http://dirac.cnrs-orleans.fr/plone/software/scientificpython`
17 `http://snake.com`

Résumé. Déterminer avec certitude si un protocole donné est sécurisé ou non est résté longtemps un défi. Le développement de techniques formelles permettant de vérifier une large gamme de propriétés de sécurité est un apport important pour relever ce défi. Ce document contribue à l'élaboration de telles techniques par la modelisation de protocole de sécurité en utilisant une algèbre de réseaux de Petri coloré appelée ABCD et réduit le temps de vérification des protocoles à l'aide de calculs distribués. Nous exploitons la nature "bien structurée" des protocoles de sécurité et la faison correspondre à un modèle de calcul parallèle appelé BSP. Cette structure des protocoles est exploitée pour partitionner l'espace des états et réduire les transitions inter-machines tout en augmentant la localité du calcul. Dans le même temps, le modèle BSP permet de simplifier la détection de la terminaison de l'algorithme et l'équilibrage des charges des calculs. Nous considérons le problème de la vérification des formules LTL et CTL* sur des systèmes de transitions étiquetées (LTS) qui modélisent les protocoles de sécurité. Un prototype a été développé, testé sur différents benchmarks.

Mots clefs. Parallélisme, model checking, logique temporelle, protocols de sécurité.

Abstract. It has long been a challenge to determine conclusively whether a given protocol is secure or not. The development of formal techniques that can check various security properties is an important tool to meet this challenge. This document contributes to the development of such techniques by model security protocols using an algebra of coloured Petri net call ABCD and reduce time to checked the protocols using parallel computations. We exploit the well-structured nature of security protocols and match it to a model of parallel computation called BSP. The structure of the protocols is exploited to partition the state space and reduce cross transitions while increasing computation locality. At the same time, the BSP model allows to simplify the detection of the algorithm termination and to load balance the computations. We consider the problem of checking a LTL and CTL* formulas over labelled transition systems (LTS) that model security protocols. A prototype implementation has been developed, allowing to run benchmarks.

Keywords. Parallelism, model checking, temporal logic, security protocols.