University of Paris-East

Habilitation thesis

Discipline : Computer Science
Presented and publicly supported by

Frédéric Gava

the december 12 2012

BSP, bon à toutes les sauces

(BSP, well suited in all kinds of ways)

Application to Functional Programming, Mechanised Verification and Security Protocols

Composition of the jury

President:	Pr. Kevin Hammond	Univ. of St. Andrews
Referees:	Pr. Jaco van de Pol	Univ. of Twente
	Pr. Joaquim Gabarró	Univ. of Catalunya
	Dr. Jean-Christophe Filliâtre	Univ. of Paris-South/CNRS
Examiners:	Pr. Herbert Kuchen	Univ. of Münster
	Pr. Marco Danelutto	Univ. of Pisa
	Pr. Zhenjiang Hu	Univ. of Sokendai/Tokyo/NII
Supervisors:	Pr. Gaétan Hains	Univ. of Paris-East
	Pr. Frédéric Loulergue	Univ. of Orléans

Notations

In this document, we write [X] for a publication of the author. The number X is preceded by one of the following acronyms: R for a publication published in a international journal, C for a publication in the proceeding of an international conference, W for an international workshop and M for the doctoral thesis of the author. We also write:

From Publications 1

> *To refer to publications that have been used for writing a section.*

We note: [[X]] (where X is a number) for a link to the url of a software. Those urls appear at the end of the document. At the end of the document, the reader can also find the list of my own publications, a glossary and my "research curriculum vitae".

Remerciement/Acknowledgements

This work was carried out under the supervision of Pr. Gaétan Hains, in the Algorithmic, Complexity and Logic Laboratory (LACL), of the University of Paris-Est Créteil.

I would like to thank all the jury members for their presence. All their papers have greatly influenced my research work. I would like to thank Kevin Hammond, who agreed to be the president of this jury. I co-organized a workshop with him, which was a great experience. Jean-Christophe Filliâtre, Jaco van de Pol and Joaquim Gabarró accepted to serve as referees. I sincerely thank them for that. I also want to express my gratitude to Herbert Kuchen and Marco Danelutto, who agreed to be part of my jury. Because of the distance, Zhenjiang Hu was only there by netmeeting; thank you for still being present, in spite of the drawbacks of such methods. Frédéric Loulergue was my advisor during my doctoral thesis; I learned research with him; thank you very much.

Je tiens également à remercier tous les membres du LACL et de l'ESIAG (où j'enseigne). Encore merci à Mathieu for its help. Je lui souhaite de réussir son concours. Merci à tout les copains et à toute ma famille pour l'aide matérielle.

À Hélène, Arthur, Zoé et Lulu.

Contents

1 Introduction

As it was noted in [1]: *The scope of software in the modern world is unprecedented. Software is now woven into artifacts such as telephones, cars, and planes; it governs the infrastructure for systems in communication, air traffic control, and banking. Faulty software contributes to the unreliability of the products in which it is embedded and is a source of significant recurring costs and delivery delays. It exposes systems to malware attacks and raises the risk of catastrophic failure in critical applications.*

My own modest work in this area concerns safety of parallel programs, their verification and the attempt to apply these works for checking security protocols. First, I will try to describe the problem statement, the model of computation behind my work as well as the projects I have been involved in. And then, I will give the outline of this document.

1.1 Context of my Work: Generalities and Background

In the context of "Think Parallel or Perish"[1], parallel code should be the norm in many areas — an attempt to list all of them would certainly fail [2]. *Easy* parallel programming and *correctness* of all parallel programs would be of paramount importance, especially considering the growing number of parallel architectures (GPUs, multi-cores, *etc.* [3]) and the *cost* of conducting large-scale simulations: the losses due to faulty programs, unreliable results, unexpected messages due to the presence of an *intruder* in the network or unexpected crashing simulations.

Formal verification tools [4] that display parallel concepts are thus useful for program understanding and incisive *debugging*. With the multi-cores, GPUs and Peta-scale revolutions looming, such tools are long overdue. Given the strong heterogeneity of these massively parallel architectures and their complexity, a frontal attack of the problem is a daunting task which is unlikely to materialise.

1.1.1 Parallel Programming

To justify what otherwise is nothing but an often-cited commonplace, we have to look at the reasons why many researchers think that parallel programming (and thus verification of parallel programs) is hard [5] and, thus, why a frontal attack seems impossible:

- **Added complexity**; there is no magic method for task decomposition and data distribution; if the wrong task decomposition is chosen, one might not see any performance increase from parallel programming; the only thing that really helps is experience; and parallel programming is not yet mainstream, too many students do not have true course of parallel computing and even if there are some books available, most of them focus on numerical algorithms (even if it is the dominant field of *High Performance Computing*, HPC); I think that learning programming and algorithms should be done directly (during the first year of courses?) in a simple parallel programming language; that would also simplify the understanding of the way to move from a sequential program to a parallel one: parallelling a sequential program is not always an incremental process and automatic parallelization tools mainly work for trivial loops.
- **Parallel programming is error-prone** even for "Gurus" of parallel computing, some subtle errors can be introduced and may appear after a large number of program executions; *deadlocks* and *data-races* are the two main problems; a lack of verification of the call of MPI's collective operators mixed with asynchronous sending and multi-threading can crash the whole machine quickly [6]; furthermore, compilers are not as well-tested as their sequential counterparts (mainly due to a lack of standards and portability of the program): due to a continuous search for efficiency, the implementation of MPI [[1]] do not always meet the specifications (which are already not always very accurate) and special libraries such as BSPLIB/PUB have not been well tested on any machine,

[1] An assertion that I find still exaggerated since there is obviously no need for parallel computing everywhere.

even when they are based on MPI. Tools for the verification of the codes exist but are not much used, mainly because they are usually not as good as their sequential counterparts; most researchers on parallel computing are not accustomed to formal methods.

- **Too few powerful abstractions are available** in the languages used for parallel programming [7]. They are often on a very low level, as each communication or synchronisation operation has to be managed in every detail by the programmer; programming becomes way easier, when the programmer can rely on powerful libraries to encapsulate complex behaviours; works on *algorithmic skeletons* and *high-level patterns* is old but there is too little knowledge of innovative parallel programming languages and tools for verification of the codes. Furthermore, it is practically impossible to know when you can compose parallel codes: how to call a MPI code within an OPENMP [[2]] one?
- **Parallel codes testing is hard**; many libraries (*e.g.* MPI, OPENMP, *etc.*) allow you to test your parallel programs on your personal computer by emulating parallelism using OS processes; but not all interleaving can be tested in this way and some combinations that generate a deadlock may be missed; parallel programming is known to be notoriously difficult, as well as tracking down compiler-introduced bugs. Testing and debugging both is usually a nightmare.

Our conclusion to why "parallel programming is hard" is that when every detail of parallelism is left to the programmer, program's complexity becomes excessive, and a large fraction of code deals with purely technical issues. A solution proposed in this document is the use of a *functional programming* language with structured parallelism. Functional languages [8] have the advantages to be high-order, safe and composing codes friendly. That is the case of structured parallelism which composes patterns of communications/computations.

1.1.2 Correctness and Verification of Parallel Programs

If parallel programs writing can be a hard task, formally proving their correctness still has a higher level of complexity. But HPC is used in many (all?) works of science and engineering. For the long term viability of this area, it is absolutely essential that we have verification tools that help application developers to gain confidence in their software. Unfortunately, the current situation is far from these ideals.

First, it is far more complex than sequential programming. Second, the mainstream in software engineering has long ignored this, neglecting alternative paradigms and conceptual work. To our knowledge, methods and tools for modelling parallel executions explicitly are far from standard in the industry yet. There is a pressing need to investigate and establish new platforms in the mainstream, suitable for the correction of parallel programs. I think that parallel programming needs more conceptual understanding.

Verification is the *rigorous* demonstration of the correctness of software with respect to a specification of its intended behaviour. For parallel programs, this lack of "understanding" makes the "rigorous" requirement difficult to satisfy. In addition to the problems of parallel programming described above, correctness of parallel programs is more sensitive to the lack of standards. If a method for the verification of programs of a given sequential programming language works well (to the opinion of the community), we can see that its adaptation to a different language is "quickly" done.

But HPC programmers and students generally simply learn C/FORTRAN+MPI. Then, they spend most of their time focusing on their application-area sciences and code correctness is often not of primary interest for them. Simple tools are thus needed to check their codes. Different solutions exist, as theorem proving and refinement "à la B" or abstract interpretation, model-checking and symbolic execution. The formers can be fully general but require interactive input from the users whereas the latter are still only applicable to a limited class of programs, but they have the advantage to be essentially automatic.

Another solution to prove functional correctness of programs or at least ensure some properties (*e.g.* deadlock free, determinism, *etc.*) is adding *assertions* in the code. One can speak of *annotated* programs and it is the traditional method for deductive verification. Assertions are important for developing reliable programs since they are used to specify a correct behaviour. They can be checked at runtime, or verified statically using a number of different techniques. Most importantly, they are easy to use, since they do not require a developer to learn much beyond the expression syntax of the programming language. It is not an automatic method but, in practise, less work is needed for it than with theorem proving. It merges both advantages of the above methods, to be partially automatic and not limited to some bounded problems such as model-checking: assertions can be introduced by hand or by automatic tools, and the generated conditions for proving the correctness can be left to automatic theorem provers.

If all these methods (realised not only as prototype tools but truly industrial applications) are now well defined for sequential programming they are less defined in HPC world. Tools for deductive verification of parallel (distributed) programs are virtually nonexistent. As for parallel programming, I think it is due

to a lack of structured parallelism: by focusing on general and concurrent parallelism, we do not define dedicated and efficient tools for structured parallelism (*e.g.* composition of patterns of communications) which is in practise, a common approach of HPC users [6, 7, 9].

Finally, *mechanised semantics* of programs and mechanised correctness proof of programs give a high confidence to their results: it is possible to verify the proof by hand; however, this is a long, tedious and error-prone process. The proofs involved in verification are very detailed and often just boring routine work. When proofs are done by hand, one tends to spend too much time checking rather simple proof steps over and over again. Therefore it is desirable to get the "help" from computers that automate the process and ensure that no mistakes are made. This is true for sequential computing, but it is even more true for parallel programs, that are more complex and have more reasons to be faulty.

1.1.3 Security Protocols

Even if we can "easily" design and verify parallel programs, in practice, few people have physical access to massively parallel machines. To take advantage of such resources, in the *cloud computing* approach, programs are sent by programmers and data stay on the servers. As our world is strongly dependent on distributed data communications, the design of secure infrastructures is thus a crucial task. Distributed systems and networks are becoming increasingly important, as a consequence, most of the services that characterise modern societies are based on these technologies. Secure communication among agents over a network is therefore a great deal for research.

Security protocols (*i.e.* cryptographic protocols) are communication protocols that use *cryptography* to achieve security goals such as secrecy, authentication or fairness in the presence of adversaries. It has long been a challenge to determine conclusively whether a given protocol is secure or not [10]. Designing security protocols is complex and often error prone [11]. *Attacks* exploit *weaknesses* in the protocol that are due to the complex and unexpected *interleaving* of different protocol sessions generated by an *intruder* (malicious) which *resides in the network*. For example, the famous "man-the-middle" attack of the Needham-Schroeder (NS) public key protocol. The intruder is assumed to have *complete network control* and to be powerful enough to perform potentially *dangerous actions* such as intercepting messages flowing over the network, or replacing them by new ones using the knowledge he has previously gained [12]. Unfortunately, the question of whether a protocol achieves its security requirements or not is, in the general case, undecidable [13, 14] or NP-complet in case of bounded number of agents [283].

Even if protocols should theoretically be checked under a unbounded number of concurrent protocol executions, violating their security requirements often exploit only a small number of sessions (an execution of an instance of the protocol) and agents. For these reasons, it is in many cases of interest sufficient to consider a finite number of sessions (in which each agent performs a fixed number of steps) for *finding flaws* — and not to prove the protocol. Formal methods offer a promising approach for automated security analysis of protocols [15]: the intuitive notions are translated into formal specifications, which is essential for a careful design and analysis. The development of formal techniques that can check various security properties is an important tool to meet this challenge [16]. Finding logical flaws by checking the validity of logical formula over the finite execution of security protocols is the *model-checking* approach for the problem of security protocols [17–19].

The greatest problem with explicit model checking security protocols is state explosion: the fact that the number of states (different configurations of the execution of the agents evolving in the protocol) typically grows dramatically with the number of agents. State space construction may be very consuming both in terms of memory and execution time: this is the so-called *state explosion problem*. The construction of large discrete state spaces is a computationally intensive activity with extreme memory demands, highly irregular behaviour, and poor locality of references. This is especially true when complex data-structures are used in the model. For example, the knowledge of the intruder is a set of values. Because the state-space construction can cause memory crashing on single or multiple processor systems, it has led considering exploit the larger memory space available in distributed systems [20].

Parallelling the state space construction on several machines is thus done in order to benefit from each machine's complete storage and computing resources. This allows to reduce both the amount of memory needed on each machine and the overall execution time.

To have efficient parallel algorithms for the state space construction, it is common to have the following requirements. First, how states are distributed across the processors must be computed quickly. Second, the successor function (of a state) must be defined so that successors states are likely mapped to the same processor as its predecessor; otherwise the computation will be overwhelmed by inter-processor communications (the so-called *cross transitions*) which obviously implies a drop of the computation

locality and thus of the performances. Third, balancing the workload is obviously needed [21] in order to fully profit from available computational power and to achieve the expected speedup. In the case of state space construction, the problem is hampered by the fact that future size and structure of the undiscovered portion of the state space are unknown and cannot be predicted in general.

Exploiting the well-structured nature of security protocols is a solution to simplify the writing of an efficient algorithm for computing the state space of finite scenario. The structure of the protocols can be exploited to partition the state space, to reduce cross transitions while increasing computation locality and to load balance the computations.

1.2 The BSP Model of Computation

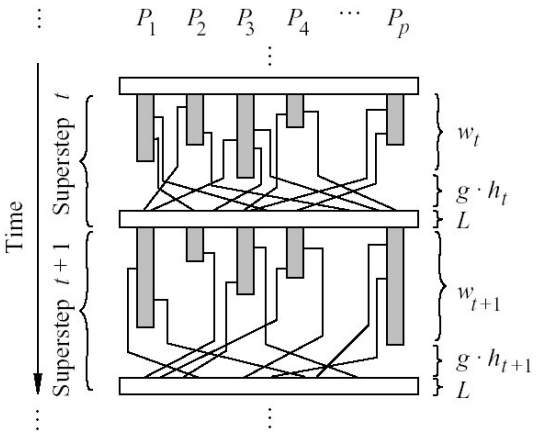
Honor to whom Honor is due. First of all, since it is the title of my habilitation, I present the BSP model of computation which is behind a large part of my work. This model of computation, its (functional) programming and some examples of applications propose solutions to the aforementioned problems. Those solutions will be presented in the following chapters.

1.2.1 The Model

The Bulk-Synchronous Parallel (BSP) model is a *bridging model* [22] between abstract execution and concrete parallel systems and was introduce by Valiant in [23] and much developed by McColl *et al.* [24]. Its initial assumptions is having portable and scalable performance prediction for parallel programs. Without dealing with low-level details of parallel architectures, the programmer can focus on algorithm design — complexity, correctness, *etc.* A nice introduction for its "philosophy" can be found in [24] and a complete book of numerical algorithms is [25]. Another presentation can be found in [26].

A BSP computer has three components: (1) a homogeneous set of uniform processor-memory pairs; (2) a communication network allowing inter processor delivery of messages; (3) a global synchronisation unit which executes collective requests for a *synchronisation 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 sub-part of the shared memory (which is then "private") and communications could be performed using a dedicated part of the shared memory. Moreover the synchronisation unit is very rarely a hardware but rather a software [27]. Supercomputers, clusters of PC s [25], multi-cores [28, 29] and GPUs [30], *etc.* can be thus considered as BSP computers.



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 left): (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.

The performance of the BSP machine is characterised by 4 parameters: (1) the local processing speed $\mathbf{r}$; (2) the number of processor $\mathbf{p}$; (3) the time $\mathbf{L}$ required for a barrier; (4) and the time $\mathbf{g}$ 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 [25]. 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:

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

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

The total cost (execution time) of a BSP program is the sum of its super-steps's costs.

1.2.2 Advantages and Disadvantages

As stated in [31]: "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 *structured* 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 informations[2]. This execution policy has two main advantages. First, it removes non-determinism and guarantees the absence of deadlocks. This is also merely the most visible aspects of a parallel model that shifts the responsibility for timing and synchronisation issues from the applications to the communications library[3]. 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 at runtime 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 livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance [24].

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 separate program phase permits a global optimisation of the data exchange by the communications library. Moreover it is easy to measure during the execution of a BSP program, time speeding to communicate and to synchronise by just adding chronos before and after the primitive of synchronisation. This facility is mainly use to compare different algorithms.

Since BSP programs are portable and cost estimate features power consumption, they can enjoy cloud-computing [33]: we can imagine a scheduler server that distributes the BSP programs depending on the BSP cost of programs to optimise power consumption and the network.

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 realise that theoretical advantage, especially when the program is to be used on a wide range of machines [6, 7].

Last advantage of BSP is that it greatly facilitates debugging. The computations going on during a super-step are completely independent and can thus be debugged independently. This facility will be used here to formally prove the correctness of our algorithms. This simplicity (for programming, debugging and proof) and yet efficiency of the BSP model makes also it a good framework for teaching: few hours are needed to teach BSP programming and algorithms.

All this capacities are 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, programmers 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 paradigm (also known as farm of processes) even if it is possible to have not too inefficient BSP programs from these schemes as we will see latter in this document. Thus, some parallel computations and optimisations would never be BSP [34]. This is the drawback of all the restricted models of computations as well.

Notice that there exist other (bridging) models of parallel computations. The most known is certainly "logp" [35, 36] but we can also cite D-BSP, E-BSP, CLUMPS, QSM, *etc.* A comprehensive list can be found in [37]. Except logp which has also been used to optimise some programs [38, 39] and MPI's collective operations implementations [40], only BSP is used and admitted. The BSP model has also been used with success in a wide variety of problems such scientific computing [25, 41–46], parallel data-structure [47, 48], genetic algorithms [49] and genetic programming [50], neural network [51], parallel data-bases [52–54],

[2] For performances issue, a BSP library can send messages during the computation phase of a super-step, but it is hidden to programmers.

[3] BSP libraries are generally implemented using MPI [32] or low level routines of the given specifics architectures.

constraints solver [55], graphs [45,56–58], geometry [59], string search [42,60,61], implementation of tree skeletons [62], search engine (queries to textual databases) [63], 3-SAT solver [64], algebra [65–67], discrete event simulation [68], bio-computing [61,69], scheduling threads [70], multi-agent services [71], image processing [30,72], *etc.* BSP was adopted because it represents a common model for writing successful parallel programs that exhibit phases-based computational behaviours [2].

1.3 Projects

During this initial part of my research career, I have been involved in different research projects that allowed me to have national and international collaborations. Moreover, the exchange of ideas has been particularly successful in the workshops including those I have organised — described in the "Research Resume" at the end of this document.

Project ACI CaraML "CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL" (2002–2005)

I involve in this project during my doctoral work. The CARAML project [[3]] aimed at implementing libraries for high-performance computing around the OCAML language. My work was to study (using operational semantics) and adapt the BSML language for globalised computations (Grid-computing). BSML is a BSP extension of the general-purpose programming language OCAML— mainly for its functional part. The extension of BSML for "grid-computing" has been done by first design an asynchronous version of BSML (call MSPML) and then by mixing BSML and MSPML: BSML on traditional parallel architectures (mainly clusters) and MSPML as a coordination language of these clusters [R5]. This is not presented here.

During my doctoral work, I also worked on machine-checked verification of BSML programs (using the COQ theorem prover) [R8], imperative features inside BSML (semantics and implementation) [C18] [R4] [R2] and applicative libraries [R3] [C7].

project ACI ProPac "Programmation paralléle certifiée" (2004–2007)

The PROPAC project [[4]] aimed at providing tools for the verification of parallel (mainly BSML) BSP programs. It succeeds to followed my work on verification of BSML programs.

The first part of the project was to increase the execution safety of BSML: handling parallel exceptions and design (with an implemented prototype) a type system were the subject of the doctoral thesis of Louis Gesbert. A machine-checked implementation of data-parallel skeletons (in BSML +COQ) has been done as a certified applicative case for the project.

The second part was to study how to prove correctness of BSP iterative programs. An extension of the deductive verification tool WHY for BSP computing was a proposed solution of the project. This was the subject of the doctoral thesis of Jean Fortin. The third part was to formalised and prove BSP costs in BSML or in iterative BSP programs. The goal is to have both correctness of results and of "virtual execution times" (step counts) of the programs.

The last part is the mechanised verification of compilers, from a kernel language to an abstract machine.

Project ANR Spreads (2007–2010)

The SPREADS [[5]] (for *Safe* P2P *REaliable Architecture for Data Storage*) project wan an ANR national project that aimed to study and design a highly dynamic secure P2P storage systems on large scale networks — Internet. It was a common research project between UBISTORAGE, I3S/INRIA/MASCOTTE, EURECOM/NSTEAM, LIP6/INRIA/REGAL and LACL. Compared to other popular peer-to-peer data sharing systems, the ones desired in this project must ensure two security properties: (1) reliability, the system must guarantee that the data will never get lost; (2) confidentiality, nobody except the data owner should be able to recover the data.

For this project, the work of the LACL members was to model (with high-level Petri nets), model-check and simulate and evaluate the quality of service of a this peer-to-peer storage system in the context of an intruder that resides in the network. These models and simulations will be performed on parallel platforms. Initially, the subject of the thesis of Michael Guedj was to design parallel algorithms for model-checking our own algebra of high-level Petri nets. His thesis was naturally diverted to the modelling of security protocols needed for this project.

Note that to have greater confidence in our results (model-checking algorithms for security properties), we considered using the tools (notably on correctness of BSP algorithms) of the PROPAC project.

1.4 Outline

This document is not organised in chronological order, but in three chapters that correspond to the three doctoral theses I co-supervised[4]. Two of the theses were done under the above projects.

First, the doctoral thesis of Louis Gesbert (co-directed with Pr. Frédéric Loulergue, LIFO, University of Orleans) [73], funded by the ACI PROPAC, was on the subject of safety of functional programming. Second, the doctoral thesis of Jean Fortin was about deductive verification of BSP algorithms; its subject is a matter of concern of the PROPAC project. And third, the doctoral thesis of Michael Guedj [74], co-directed with Pr. Franck Pommereau (IBISC, University of Evry) and funded by the VEHICULAR project and under the ANR SPREADS project, was on design of BSP algorithms for model-check security protocols.

The sequence of chapters follows the order of the projects I've been involving: first the CARAML and PROPAC projects about safety of parallel functional programming; then the continuity of the PROPAC project about deductive verification of parallel iterative programs; and to finish the design of (proved correct) algorithms for the verification of security protocols. In fact, some results of the PROPAC project are currently used in the SPREADS project.

Each chapter ends with related works and what we have learnt from these works.

Chapter 2: Functional BSP programming

In this chapter, we describe our contribution to functional parallel programming.

First, we give an introduction to the main ideas of our studied language called BSML. Second, we describe the language: we give its history, its model of execution and an informal semantic of its parallel primitives. Each primitives is illustrated with simple examples. Third, we give some extensions, such as a primitive of parallel composition or a handling of parallel exceptions. Fourth, we highlight how some BSML programs can crash the whole parallel machine. A Hindley/Milner type system (with effects and constraints) is then presented and we show how it prevents from this kind of program.

Fifth, as an example of how BSML can be used to program higher-order parallel patterns, we give a BSML implementation of two kinds of algorithmic skeletons: data-flow and data-parallel. It is interesting to note that the first implementation massively use the primitive of parallel composition. Finally, since BSML is a purely functional extension of ML, we emphasise how this property can be used to handle BSML primitives within a theorem prover (here COQ): an axiomatization of the BSML primitives in COQ, ables to benefit from the extraction of COQ for having certified parallel programs.

Chapter 3: Deductive Verification of BSP Algorithms

We now turn to the deductive verification of iterative BSP algorithms. These algorithms contain traditional BSP primitives (those of most BSP libraries of programming languages as C or JAVA have) mainly for sending messages and global synchronisations — the well-known barrier. Verification means proving formally that a program satisfies its specification, which is written in a logical language. Some techniques are based on testing some appropriate executions, some others on checking all possible executions or checking dynamically the program during its execution. These techniques help to find bugs, but do not provide a reliable proof of correctness. It is indeed impossible to verify functional specifications exhaustively by such methods, simply because the number of possible executions is too large or even infinite. We need formal verification methods to prove that a program always satisfies its specification. The approach used in this work is based on the method initiated by Hoare for sequential programs and extended here to BSP programs.

We thus focus on correctness of BSP algorithms, that is, characterise the precise meaning of what the parallel algorithm is doing and what we expect from it. This means proving formally that a program performs its intended task for any possible input. Traditional pre- post- conditions and loop invariants (the famous Hoare triples) will be used. The technique is based on proving that a given annotated algorithm fits its logical specifications using a generation of logical conditions called goals. The generation of those goals is generally based on the famous Dijkstra's weakest precondition[5].

[4] But there are dependencies of the results of the doctoral theses.

[5] Whereas Hoare-like techniques are based on proving that a given program together with its specification can be derived from a system of axioms and inference rules which are syntax oriented.

Our work is a BSP extension of a deductive tool called WHY. This tool takes annotated algorithms in a programming language called WHY-ML[6] and extracts goals to pass them to provers. Those goals ensure the correctness of some properties of the program.

Writing such a tool is a tremendous amounts of works which should be left to the field experts. The main idea of our work is to simulate the parallelism by using a transformation of the parallel code into a pure sequential one. Therefore, we now use a "well defined" verification tool of sequential algorithms as a back-end for verification of parallel algorithms. In this chapter, we first describe the BSP-WHY-ML language and the transformation of BSP-WHY-ML expressions into WHY-ML ones. We also give some simple examples of correctness of BSP algorithms in order to show its relative effectiveness.

Finally, we describe two mechanised semantics of this language: (1) a big step semantics as a formal specification of the language, that can also be used in the future for proving the above transformation; (2) a small step semantics that can be used for compiler mechanised verification. Given these semantics for a kernel language that is WHY-ML is a first step to realise the underlying difficulties before working on real programming languages like C or JAVA.

Chapter 4: BSP Algorithms for the Verification of Security Protocols

We recall that security 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. Moreover, in case of parallel computations over cloud/grid-computing environments, that is giving a parallel program and data to remote computers, user should be well identified, assured of the secret of his own data and assured of the fairness of the environment.

As said above, designing secure protocols is a challenging problem [10, 75]. In spite of their apparent simplicity, they are notoriously error-prone. "Surprisingly", severe attacks can be conducted even without breaking cryptography, but only by exploiting weaknesses in the protocols themselves, for instance by carrying out man-in-the-middle attacks (an attacker plays off one protocol participant against another), or replay attacks — messages from one session 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.

In this chapter, we exploit the well-structured nature of security protocols and match it to the structured BSP model. This allows 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. The tool for deductive verification of the previous chapter will also be used to check the correctness of our own parallel algorithms, in order to increase the confidence we can have of these security results.

We also consider the problem of checking logical formula (LTL and CTL*) over labelled transition systems (LTS) that model security protocols. Checking a formula over a protocol is not new [76] but it has the advantage over dedicated tools for protocols to be easily extensible to non standard behaviour of honest principals and to check some security goals that cannot be expressed as reachable properties, *e.g.*, fair exchange. By using the above mentioned partition of the data, we were able to design efficient parallel algorithms for LTL and CTL* checking.

[6] WHY-ML is language with a syntax close to ML; but it is only a first-order language.

2 Functional BSP Programming

This chapter resumes the work of [R1], [R2], [R3], [R4], [R8] and [C4], [C7], [C11], [C15] and [W1], [W3] and [N1]. This work was also done within the framework of the Doctoral thesis of Louis Gesbert [73] (defence in March 2009) co-directed with Pr. Frédéric Loulergue and funded by the PROPAC *project.*

We recall that writing parallel programs is known to be notoriously difficult. Often programmers do not want to reason about message-passing algorithms and only want to combine existing high-level patterns to produce their parallel programs. The Bulk Synchronous Parallel ML (BSML) language is a parallel extension of ML— a family of functional programming languages. BSML is an attempt at providing the right balance between the two opposite approaches of parallel programming: low-level (and subject to concurrency issues), and high-level — with loss of flexibility and efficiency.

In this chapter, we present the BSML language and what has been done by the author about BSML. We present the language itself and some of its extensions. We also present some examples of applications and benchmarks.

Contents

2.1 Functional BSP programming

2.1.1 Generalities

The design of a parallel language is a tradeoff between the possibility for the programmer to control parallel aspects necessary for predictable efficiency (but which make programs more difficult to write, to prove and to port) and the abstraction of such features which are necessary to make parallel programming easier — but which hampers efficiency and performance prediction.

(a) "Philosophy"

Our contribution is BSML, an extension of ML to code BSP algorithms which combines the high degree of *abstraction* of ML (without poor performances because very often, ML programs are as efficient as C ones) with the *scalable* and *predictable* performances of BSP. It aims at providing a good balance between low-level programming and high-level one. In the former, we find libraries such as MPI generally used with FORTRAN or C; these approaches are unsafe and leave the programmer responsible for deadlock or non-determinism issues[1]. In the latter stand traditional algorithmic skeletons [77] where programs are safe but limited to a restricted set of algorithms.

BSML uses a *small set of primitives* which are currently implemented as a parallel library [[6]] for the ML programming language OBJECTIVE CAML — OCAML [[7]]. Using a safe high-level language as ML to program BSP algorithms allows performance, scalability and *expressiveness*. This choice was made among the different variants of ML available mainly for a reason of efficiency, since we target high-performance computation. Other reasons include the amount of libraries available and the tools provided. We plan a full language implementation by generating adequate OCAML code.

BSML follows the BSP paradigm to structure the computation and communication between the processors in a data-parallel fashion. All communications in BSML are collective (require all processes) and deadlocks are avoided by a strict distinction between local and global computation.

BSML is based on the BSP structured model of parallelism and is universal for this model: any BSP algorithm could be written using BSML. Other structured parallelism approaches such as algorithmic skeletons first define a set of parallel patterns, the skeletons, and the model of parallelism is then derived from this set. When the set is not fit to a particular algorithm, it may be extended: This requires of course of lot of work for the library implementer and does not help the user of the library who has to choose among a bigger set of skeletons.

On the contrary, BSML offers a very small set of parallel primitives over a parallel data structure, called parallel vector: two functions to manipulate this structure as well as four constants to access the BSP parameters of the underlying architecture. By comparison, the standard BSPLIB [[8]] [78] library for BSP programming in C offers about fifteen primitives, and MPI more than a hundred.

Another important feature of BSML is its *determinist* (except when using random primitives) semantics [M1]: what ever the order of execution of the processors, the final value would the same. This feature is uncommon in the world of parallel languages [79] — even if it is a desirable properties. For many computational problems, there is no inherent non-determinism in the problem statement. Determinist is convenient for safety: no deadlock nor data-race should occur. It allows the programmer to reason on its algorithms and their optimisations without dealing with these problems of safety: the programmer reasons "as" for a sequential computation which is generally simpler.

(b) History of BSML

The first work on BSML was done in [80] to define the BSλ-calculus which aims to be the core-language of BSML. Distinct syntax are used to separate global terms (parallel) and local ones (purely sequential). That allows neither an easy programming nor polymorphism in term. A first sequential implementation was also given as well as a way to have a parallel implementation. A semantics study was done in [81] which was improve in [M1] by the use of an abstract machine and the current parallel primitives of BSML. In [M1] big-step, small-step and distributed semantics have been proved equivalent and confluent.

In [82], a presentation and benchmarks of the first parallel implementation of BSML as a library was done using the BSPLIB, a library for BSP programming in C. [83] presents a first virtual machine for BSML. But the set of instructions needs for parallelism is too large to be used in a realistic implementation. The first MPI implementation of BSML was done in [84]. [85] presents a first adaptation of BSML for C ++.

[1] These properties are justified for concurrent computations but clearly not for parallel algorithms.

Another parallel machine was studied in [C23] which is the basis of the modular implementation of [C15]: it defines a small number of instructions that are needed for having a parallel implementation of BSML. MPI, PUB [[9]], TCP/IP, SHM and PVM [[10]] implementations were done and benchmarked — currently, only the MPI version is stable and efficient.

The first use of COQ for correctness of BSML programs appears in [R8]. The first type system appears in [R2]. Both were improve in [73]. [86] presents systematic development of correct BSML programs using the theorem prover COQ. The authors refine a BSML program from a specification in COQ, using a set of rewriting rules of the Bird-Meertens formalism [87] — mainly on a homomorphic skeleton which is specific to BSP computing. The correctness of BSML programs using COQ was also improved in [88].

[89] presents a first composition primitive for BSML. It was improved in [90] and the first effective implementation was done in [C12]. The future version should use the new implementation design in [R1].

2.1.2 Model of Execution

BSML comes, as OCAML, with three modes of compilation/evaluation:

- a byte-code compiler (a set of scripts calling the OCAML byte-code compiler with the appropriate BSML modules, in a similar way `mpicc` is not a full compiler; there are several scripts as there are several available implementations of BSML depending on the underlying low-level communication used: TCP/IP, MPI, *etc.* or sequential);
- a native code compiler (also a set of scripts);
- an interactive loop also called top-level.

This interactive loop is a sequential simulator for BSML. Thanks to BSML semantics's properties it is ensured that the parallel and the sequential implementations behave the same [81] [M1].

The core syntax of BSML is that of OCAML— with few restrictions. BSML programs can mostly be read as OCAML ones, in particular, the execution order should not seem unexpected to a programmer used to OCAML, even though the program is parallel. Moreover, most normal OCAML programs can be considered as BSML programs that do not make use of parallelism: the programs are executed sequentially on each processor of the parallel machine and return their results normally. That allows the parallelisation to be done incrementally from a sequential program.

Few entry points are needed for parallelism. BSML is based on a data-type called *parallel vector* which, among all OCAML types, enables parallelism. A parallel vector has type 'a par and embeds $\mathbf{p}$ values of any type 'a at each of the $\mathbf{p}$ different processors in the parallel machine. The $\mathbf{p}$ processors are labelled with integers from 0 to $\mathbf{p}-1$ which we call *pid* (Processor Identifier) of the processors. The nesting of parallel vectors is not allowed — see Section 2.3 for a discussion and a type system that prevents this forbidden use of vectors.

We use the following notation to describe a parallel vector:$\langle x_0, x_1, \ldots, x_{\mathbf{p}-1}\rangle$ where $\mathbf{p}$ is the (constant) number of processors throughout the execution of the program. It can be accessed in BSML using the integer constant **bsp_p** — the other BSP parameters are also accessible as float values through constants **bsp_g**, **bsp_l** and **bsp_r**. We distinguish a parallel vector from an usual array of size $\mathbf{p}$ because the different values, that will be called *local*, are blind from each other; it is only possible to access the local value x_i in two cases: locally, on processor i (by the use of a specific syntax), or after some communications. In this way, the vector $\langle x_0, x_1, \ldots, x_{\mathbf{p}-1}\rangle$ holds the value x_i at processor i.

These restrictions are inherent to distributed memory parallelism. This makes parallelism explicit and programs more readable. Since a BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution that take place will be needed:

- **Replicated** execution is the default. Code that does not involve BSML's parallel vectors is run by the parallel machine as it would be by a single processor. Replicated code is executed at the same time by every processor, and leads to the same result everywhere.
- **Local** execution is what happens inside parallel vectors, on each of their components: The processor uses its local data to do computation that may be different from the other's.
- **Global** execution concerns the set of all processors together, but as a whole and not as a single processor. Typical example is the use of communication primitives.

The distinction between local and replicated is strict. Hence, the replicated code can not depend on local information, and remains replicated. We say that we lose replicated consistency if an expression outside of a parallel vector holds different values at different processors.

primitive	type	informal semantics
$\ll$ e $\gg$	t par (if $e : t$)	$\langle e, \ldots, e \rangle$
pid (within a vector)	int	i on processor i
v (within a vector)	t (if $v : t$ par)	v_i on processor i (if $v = \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$)
`proj`	'apar→ (int→ 'a)	$\langle x_0, \ldots, x_{p-1} \rangle \mapsto$ (`fun` $i \to x_i$)
`put`	(int→ 'a)par→ (int→ 'a)par	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle$(`fun` $i \to f_i$ 0), ..., (`fun` $i \to f_i$ ($\mathbf{p}-1$))$\rangle$
`super`	(unit→ 'a)→ (unit→ 'a)→ 'a*'b	$f_a \mapsto f_b \mapsto (f_a\ (), f_b\ ())$

Figure 2.1. Summary of the BSML primitives.

2.1.3 Parallel Primitives and their Informal Semantics

Parallel vectors are handled through the use of different communications primitives that constitute the core of BSML. Their implementation relies either on MPI, PUB [91] or on the TCP/IP functions provided by OCAML. Figure 2.1 subsumes the use of these primitives.

(a) Parallel Primitives

Informally, primitives works as follows. Let $\ll$ x $\gg$ be the vector holding x everywhere — on each processor. The $\ll \gg$ indicates that we enter a local section and pass to the local level. Replicated information is available inside the vector. For example, using the toplevel:

```
Bulk Synchronous Parallel ML version 0.5
  The BSP Machine has 4 processors
  o BSP parameters g = 20.3 flops/word
  o BSP parameters L = 4571. flops
  o BSP parameters r = 498952227. flops/s

# open Bsml;;
# let vec1 = ≪ "HDR" ≫ ;;
val vec1 : string Bsml.par = <"HDR", "HDR", "HDR", "HDR">
```

The # symbol is the prompt that invites the user to enter an expression to be evaluated. The top-level then gives an answer of the form: name of the defined value, type and pretty-printing of the value. The values of the BSP parameters are measured values of a quad-core i7 machine.

Now, to access local information, we add the syntax x to open the vector x and get the local value it contains, which can obviously be used only within local sections. The local *pid* can be accessed with $**pid**$. For example:

```
let vec2= ≪ $vec1$^"␣on␣proc␣"^(string_of_int $pid$) ≫ ;;
val vec2 : string Bsml.par = <"HDR␣on␣proc␣0", "HDR␣on␣proc␣1", "HDR␣on␣proc␣2", "HDR␣on␣proc␣3">
```

The **proj** primitive is the only way to extract a local value from a vector. Given a parallel vector, it returns a function such that applied to the *pid* of a processor, it returns the value of the vector at this processor. **proj** performs communications to make local results available globally within the returned function. Hence it establishes a meeting point for all processors and, in BSP terms, ends the current super-step. **proj** is often used at the end of a parallel computation to gather the computed results. For example, if we want to convert a parallel vector into a list, we write:

```
# let list_of_par vec = List.map (proj vec) procs;;
val list_of_par : 'a Bsml.par → 'a list = <fun>
# list_of_par ≪ $pid$ ≫ ;;
- : int list = [0; 1; 2; 3]
```

where procs is the list of *pid*s [0; 1; ... ; **bsp_p**−1].

Note the choice of functions of type (int → 'a) in **proj**. Arrays of size **p** or lists could have been chosen instead, but the interface is more functional and generic this way. Furthermore, as seen in the examples, the conversion between one style and the other is easy. Internally, our implementation relies on arrays.

The **put** primitive is the comprehensive communication primitive. It allows any local value to be transferred to any other processor. As such, it is more flexible than **proj**. It is as well synchronous, and ends the current super-step. The parameter of **put** is a vector that, at each processor, holds a function of type (int → 'a) returning the data to be sent to processor j when applied to j. The canonical use of **put** is: (**put** $\ll$ **fun** sendto → e($**pid**$, sendto, x) $\gg$) where expression e computes (or usually, selects) the data that should be sent depending on sender to sendto. The result of **put** is another vector of functions: At a processor j the function, when applied to i, yields the value *received from* processor i

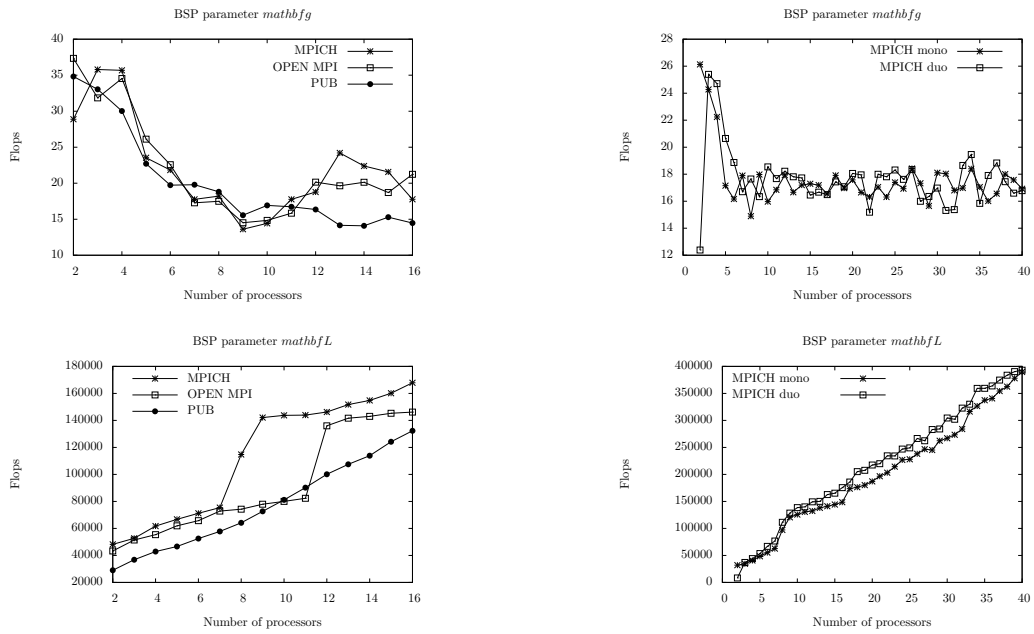


Figure 2.2. BSP parameters **g** and **L** of our machines LACL1 (left) and LACL2 (right).

by processor j. Some values, as the empty list or the first constructor without parameters in a sum type, are considered to mean "no message". For example, a total_exchange could be written:

```
# let total_exchange vec =
  let msg = put ≪ fun dst → $vec$≫ in
  ≪ List.map $msg$ procs ≫
val total_exchange : 'a Bsml.par → 'a list Bsml.par = <fun>
# total_exchange ≪ $pid$ ≫ ;;
− : int list Bsml.par = <[0;1;2;3], [0;1;2;3], [0;1;2;3], [0;1;2;3]>
```

Note that the "same" function is list_of_par because replicated values are the same on every processors. The BSP cost on this function is $(\mathbf{p}-1)\times s\times \mathbf{g}+\mathbf{L}$ where s is the size of the bigger value send by a processor.

(b) Modular Implementation

Using the work of [C23], we find that only few operations (instructions in the abstract parallel machine) are really needed for implementing the BSML primitives. Mainly, given the *pid* of the processors and an instruction that performs a (synchronous version) MPI's alltoall collective operation. This allows a modular implementation since only these instructions need to be implemented, the rest follows.

In [C15], we have used this fact to implement these instructions using different libraries of parallel computing. We also perform some benchmarks: PUB, a BSP library for C, is the most efficient whereas MPI is the more stable.

2.1.4 Benchmarking of the BSP Parameters

One of the main advantages of the BSP model is its cost model: it is quite simple and yet accurate. We used a BSML version of a program of [25] to benchmarks and determines the BSP parameters of two parallel machines:

- LACL1 is a cluster of 16 PC s connected through a 1 Gigabyte Ethernet network; Each PC is equipped with a 2.8 Ghz Intel® Pentium® IV CPU, with 1 GB of physical memory (RAM);
- LACL2 is a cluster of 20 PC s 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.

Each node of the clusters works with linux Ubuntu. For the LACL1 cluster, we compute these parameters for 3 libraries: MPICH, OPEN-MP and PUB [91]. We have compute only for the MPICH library for the LACL2 cluster — the machine were too used for the test of the work of Chapter 4.

Figure 2.2 summarises the timings of **L** and **g** (where **r** is 330 Mflops/s for each library of LACL1 and 220 Mflops/s on LACL2) for an increasing number of processors of LACL1 (in left) and LACL2 (in right). For LACL2, due to a dual core architecture we have perform two kinds of benchmarks: (1) distributed the processes on processors (cores) making priority to nodes (maximally spread the processes, we call this strategy "mono"); (2) making priority to cores (gather a maximum of processes on nodes) that is to say, fill up the cores of each node (call "duo").

For both machines and both strategy of LACL2, we notice that the parameter **L** is growing in a quasi-linear way. However, for the MPI libraries on LACL1, two jumps are visible. No explanation has been found yet. For parameter **g** of LACL1, it is surprising to see that it is high for a few processors and then more stable — real parameter exchange of the network. For the "duo" strategy of LACL2, both **L** and **g** parameters are good for 2 processors: this is due to the used of only one node which implied that only share-memory exchanges are performed. But when different nodes are used, the performances of the network surpass the ones of share-memory. We also notice a little bottleneck of the network cards when two cores of a single node are considering as two different machines of the BSP machine. We believe that it will be worse when more nodes are present in a machine (supposing 16, 32, 64 cores on future PC s) making the BSP model not yet accurate. We will discuss of this issue in the conclusion and future works.

Note that most of the benchmarks (and test of adequacy between true timing and predicted one using the BSP parameters) for BSML presented below have been performed using the LACL1 cluster while benchmarks of Chapter 4 (parallel model-checking of security protocols) have been done using LACL2. This is due to a change of cluster during these works.

2.1.5 Some Examples and Applications

From Publications 2

This sub-section summarises the work in [C11], [C7] and [W3].

Having a very small core of parallel operations is a great strength for the formalisation of the language. It makes the definitions clear and shorter. The primitives described in the previous section constitute the core of BSML. However, the use of the primitives can sometimes become awkward. Defining some useful libraries simplify the coding of the algorithms. We define a few functions useful for BSP computing where some are given as additional BSML libraries. They are typical examples of BSML programming.

(a) Utility Functions

It is often needed to split a data set among the processors. The following function is an example that selects a part of a list on every processor:

```
(* select_list:'a list → 'a list par *)
let select_list l = let len = List.length l
                    in ≪ cut_list l ($pid$ * len / bsp_p) (($pid$ + 1) * len / bsp_p))≫
```

where cut_list l a b returns the sub-list of the elements of l from index a (inclusive) to index b — exclusive. Now, to parallel map a function on a list (classical data-parallel skeleton [77, 92]) scattered as above is as simple as:

```
(* parmap : ('a→ 'b)→ 'a list par → 'b list par *)
let parmap f parlist = ≪ (List.map f) $parlist$≫
```

Broadcasting (from one processor to others as MPI_bcast) is also useful. A naive and direct version could be code as follow:

```
(* bcast_direct: int→ 'a par→ 'a par *)
let bcast_direct root vv =
 if not (correct_number_of_processor root)
  then failwith "Bcast: not a correct number of processor"
 else
  let rcv = put ≪ if $pid$=root then (fun dst → Some $v$) else (fun dst → None) ≫
   in ≪ noSome ($rcv$ root)≫
```

The BSP cost of this function is $(\mathbf{p}-1) \times s_{v_{root}} \times \mathbf{g} + \mathbf{L}$ where $s_{v_{root}}$ is the size of the value of processor *root*. Note that when this value is important and depending of the BSP parameters, another and more efficient algorithms could be used [25, 41] — mainly a two super-steps algorithm where in the first super-step the value is cut into small piece which are them exchange and then gather on processor in a second super-step; for sack of concision, we do no show this algorithm here.

Reducing and scanning are also important operator — both MPI's collective operators. A simple one-step reduce (having $\oplus_{k=0}^{\mathbf{p}-1} v_k$ on each processor from the parallel vector $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1} \rangle$) can be done with:

```
(* scan': int→ ('a→ 'a→ 'a)→ 'a→ 'apar→ 'apar *)
let rec scan' step op e v =
 if step >= bsp_p then v else
  let comm =
     put ≪ fun j →
               if (j mod (2*step) = 0)
                 && ($pid$ = j + step)
                then $v$
               else e ≫
   in let v' =
        ≪ if $pid$ mod (2*step) = 0
          then
           if $pid$ + step < bsp_p
            then op $v$ ($comm$ ($pid$ + step))
           else $v$
         else e ≫
  in scan' (step*2) op e v'

(* reduce: ('a→ 'a→ 'a)→ 'a→ 'a par→ 'a *)
let reduce op e v = (proj (scan' 1 op e v)) (bsp_p-1)
```

```
(* inbounds: 'a→ 'a→ 'a→ bool *)
let inbounds first last n = (n>=first)&& (n<=last)
(* mix: int→ 'a par * 'a par→ 'a par *)
let mix m (v1,v2) = ≪ if $pid$<=m then $v1$ else $v2$ ≫

(* scan_super: ('a→ 'a→ 'a)→ 'a→ 'a par→ 'a par *)
let scan_super op e vec =
 let rec scan' fst lst vec =
  if fst>=lst then vec
  else
   let mid = (fst+lst)/2 in
   let vec'= mix mid (super (fun()→ scan' fst mid vec)
                            (fun()→ scan'(mid+1) lst vec)) in
   let send = put ≪ if $pid$=mid
                    then (fun dst→ if inbounds (mid+1) lst dst
                                   then Some $vec'$
                                  else None)
                   else (fun dst→  None)≫  in
     ≪ match ($send$ mid) with
         | None → $vec'$
         | Some v → op v $vec'$ ≫
 in scan' 0 (bsp_p()-1) vec)
```

Figure 2.3. Parallel reduce computation (left) and its Divide-and-conquer version (right).

```
(* simple_reduce: ('a→ 'a→ 'a)→ 'a→ 'a par → 'a
let simple_reduce op e v = List.fold_left op e (proj_list v)
```

where e is a neutral element and op the associative operator $\oplus$.

The above reduce does not make use of parallelism. If the combination operator $\oplus$ has some cost, we may prefer to reduce in a multi-step manner (a classic logarithmic way), doing the combinations locally. This algorithm combines the values of processors i and $i + 2^n$ at processor i for every super-step n from 0 to $\lceil \log_2(\mathbf{p}) \rceil$. Figure 2.3 (left) gives the code.

The program (scan' 1) gathers data at every even processor, then at multiples of 4, 8, *etc.* Communication is the first step: the argument of `put` returns the operator's unit unless sending from processor $(2 \times step + 1) \times i$ to $2 \times step \times i$ for any i. Then, the combination is done at processor $2 \times step \times i$ using op.

(b) Implementation of Parallel Algorithms

Parallel Sorting. Our first example is the *sampling sort algorithm* (PSRS) of Schaeffer in its BSP version [45]. The PSRS algorithm proceeds as follows. First, the lists of the parallel vectors (we assume that their lengths are $\geq \mathbf{p}^3$) 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 samples. 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. It can be coded in BSML as follows:

```
(* psrs: int par → 'a list → 'a list *)
let psrs lvlengths lv =
    (* super−step 1(a): local sorting *)
  let locsort = ≪ List.sort compare $lv$ ≫ in
    (* super−step 1(b): selection of the primary samples *)
  let regsampl = ≪ extract_n bsp_p $lvlengths$ $locsort$ ≫ in
    (* super−step 2(a): total exchange of the primary samples *)
  let glosampl = List.sort compare (gather_list regsampl) in
    (* super−step 2(b): selection of the secondary samples *)
  let pivots = extract_n bsp_p (bsp_p*(bsp_p−1)) glosampl in
    (* super−step 2(c) : building the communicated lists of values *)
  let comm = ≪ slice_p $locsort$ pivots ≫ in
   (* super−step 3: sended them and merging of the received values *)
  let recv = put ≪ List.nth $comm$ ≫ in
    ≪ p_merge bsp_p (List.map $recv$ procs_list) ≫
```

where "vlengths" is the vector of list sizes, "extract_n n" extract $n-1$ elements evenly distributed; "slice_p" transform a sorted list to a list of list depending of the samplings and "p_merge" merges sorted lists. The cost of the first super-step is:

$$\frac{n}{\mathbf{p}} \times \log(\frac{n}{\mathbf{p}}) \times c_c + \frac{n}{\mathbf{p}} + (\mathbf{p} \times (\mathbf{p}+1) \times s_e) \times \mathbf{g} + \mathbf{L}$$

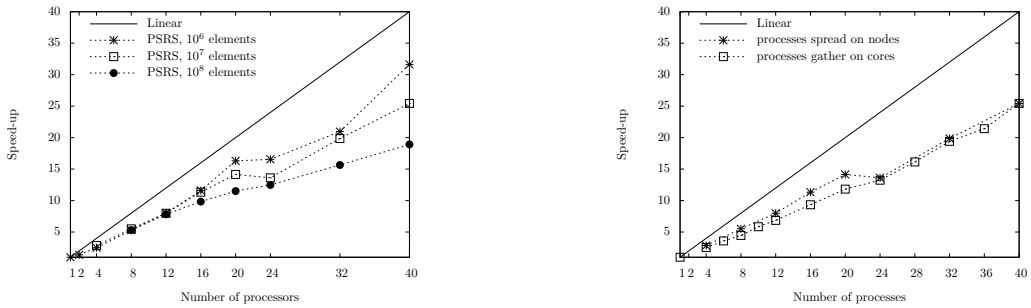


Figure 2.4. Benchmarks of the PSRS sorting: for different numbers of elements (left); for 10^7 elements and comparison of the scheme of distribution over the machine of LACL2 (right).

where c_c is the time (suppose constant) to compare two elements and s_e the size (also suppose constant) of an element. The cost of the second super-step is [45]:

$$\frac{n}{\mathbf{p}^2} \times \log(\frac{n}{\mathbf{p}^2}) \times c_c + \frac{n}{\mathbf{p}^2} + \frac{3n}{p} \times s_e \times \mathbf{p} + \mathbf{L} + \text{time}_{\text{fusion}}$$

where the time of fusion on each processor is $O(n/\mathbf{p})$. Figure 2.4 shows the performances of the PSRS algorithm for the LACL2 machine for different number of elements in the distributed list (left) and for the two mentioned strategies — spread or gather processes.

Sieve of Eratosthenes. For this problem, we give the parallel methods, BSP cost formula as well as tests of comparison between theoretical (depending on the BSP parameters) and experimental performances. This comparison shows that the BSP cost analysis would help choosing the best BSML program.

The sieve of Eratosthenes generates a list of primary numbers below a given integer n. We study three parallel methods. We generate only the integers that are not multiple of the four first prime numbers and we classically iterate only to $\sqrt{n}$. The probability of a number a to be a prime number is $\frac{1}{\log(a)}$, so we deduce a complexity of $(\sqrt{n} \times n)/\log(n)$.

Figure 2.5 gives the BSML code of the three methods. We used the following functions: elim:int list$\rightarrow$ int $\rightarrow$ int list which deletes from a list all the integers multiple of the given parameter; final_elim:int list$\rightarrow$ int list$\rightarrow$ int list iterates elim; seq_generate:int $\rightarrow$ int$\rightarrow$ int list which returns the list of integers between two bounds; and select:int $\rightarrow$ int list$\rightarrow$ int list which gives the $\sqrt{n}$th first prime numbers of a list.

Logarithmic reduce method. For our first method we use the classical parallel prefix computation. We use the divide-and-conquer BSP algorithm which used the superposition. But in our computation, the sent values are first modified by a the functionselect which select the $\sqrt{n}$th first prime numbers.

The parallel methods is thus very simple: each processor i holds the integers between $i \times \frac{n}{\mathbf{p}} + 1$ and $(i+1) \times \frac{n}{\mathbf{p}}$. Each processor computes a local sieve (the processor 0 contains thus the first prime numbers) and then our scan is applied. We then eliminate on processor i the integers that are multiple of integers of processors $i-1$, $i-2$, *etc.* We have $\log(\mathbf{p})$ super-steps where each processor sent/received at most two values (list of size max $\sqrt{n}$). The BSP cost is accordingly (where $m = \frac{n}{\mathbf{p}}$):

$$\log(\mathbf{p}) \times (\frac{\sqrt{m} \times m}{\log(m)} + 2 \times \sqrt{n} \times \mathbf{g} + \mathbf{L})$$

Direct method. It is easy to see that our initial distribution (bloc of integers) gives a bad load balancing (processor $\mathbf{p}-1$ has the bigger integers which have little probability to be prime). We will distributes integers in a cyclic way: a is given to processor i where $a \bmod \mathbf{p} = i$). The second method works as follows: each processor computes a local sieve; then integers that are less to $\sqrt{n}$ are globally exchanged; a new sieve is applied to this list of integers (thus giving prime numbers) and each processor eliminates, in its own list, integers that are multiples of this $\sqrt{n}$th first primes. The BSP cost is accordingly:

$$2 \times \frac{\sqrt{m} \times m}{\log(m)} + \frac{\sqrt{\sqrt{n}} \times \sqrt{n}}{\log(\sqrt{n})} + \sqrt{n} \times \mathbf{g} + \mathbf{L}$$

Recursive method. Our last method is based on the generation of the $\sqrt{n}$th first primes and elimination of the multiples of this list of integers. We generate this by a inductive function on n. We

```
let eratosthene_scan n =
  let p=bsp_p() in
  let listes = ≪ if $pid$=0 then seq_generate (n/p) 10
                 else seq_generate ((pid+1)*(n/p)) (pid*(n/p)+1))≫ in
  let local_eras = ≪ (local_eratosthene n) $listes$≫ in
  let scan_era = scan_super final_elim (select n) local_eras in
   ≪ if $pid$=0 then 2::3::5::7::$scan_era$ else $scan_era$ ≫

let eratosthene_direct n =
 let listes = ≪ local_generation n $pid$≫ in
 let etape1 = ≪ local_eratosthene n $listes$≫ in
 let selects = ≪ select n $etape1$≫ in
 let echanges = replicate_total_exchange selects in
 let premiers = local_eratosthene n
                  (List.fold_left (List.merge compare) [] echanges) in
 let etape2 = ≪ final_elim premiers $etape1$≫ in
  ≪ if $pid$=0 then 2::3::5::7::(premiers@$etape2$) else $etape2$ ≫

let rec eratosthene n =
  if (fin_recursion n) then ≪ (distribution $pid$) (seq_eratosthene n) ≫
  else
   let carre_n = int_of_float (sqrt (float_of_int n)) in
   let prems_distr = eratosthene carre_n in
   let listes = ≪ local_generation2 n carre_n $pid$ ≫ in
   let echanges = replicate_total_exchange prems_distr in
    let prems = (List.fold_left (List.merge compare) [] echanges) in
     ≪ (final_elim prems) $listes$ ≫
let eratosthene_rec n =
 let era=eratosthene n in ≪ if $pid$=0 then 2::3::5::7::$era$ else $era$ ≫
```

Figure 2.5. BSML code of the the parallel versions of the sieve of Eratosthenes.

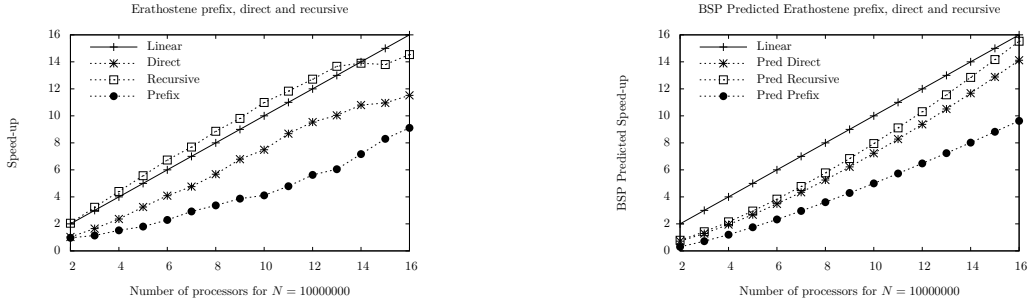


Figure 2.6. Performances (using PUB) of the sieve of Eratosthenes.

suppose that the inductive step gives the $\sqrt{n}$th first primes and we perform a total exchange on them to eliminates the non-primes. End of this induction comes from the BSP cost: we end when n is small enough so that the sequential methods is faster than the parallel one. The inductive BSP cost is accordingly:

$$\begin{aligned} \text{Cost}(n) &= \frac{\sqrt{m}\times m}{\log(m)} + \sqrt{n} \times \mathbf{g} + \mathbf{L} + \text{Cost}(\sqrt{n}) \\ \text{Cost}(n) &= \frac{\sqrt{n}\times n}{\log(n)} \qquad \text{if BSP cost} > \text{sequential complexity} \end{aligned}$$

Figure 2.6 gives the predicted and measured performances (using the PUB implementation) for the cluster LACL1. To simplify our prediction, we suppose that pattern-matching and modulo are constants in time — we measure them by timing these operations a hundred times. Size of lists of integers can be measured using the Marshal module of OCAML. Note that we obtain a super-linear acceleration for the recursive method. This is due to the fact that, using a parallel method, each processor has a smaller list of integers and thus the garbage collector of OCAML is called less often. One can notice that predicted performances using the BSP cost model are close to the measured ones.

(c) Scientific Computations

We now gives some references of the use of BSML for implementing scientific applications. In [W3] we have implemented in BSML different algorithms for solving the "heat equation problem" — variation in temperature over time in a given material. Experiments of this program were performed on the cluster (8 PC s, each with two Quad-Cores processors) of the Orléans's laboratory of computer science.

We have also implemented in [C7] a library for dense matrix operations. In this work, a matrix is

```
(* multiply_par:t→ t→ t *)
let multiply_par parA parB =
 let parC=ref mat_create neutral parA.size_row in
 let sqrt_p=sqrt_int bsp_p in
 let ni=n/sqrt_p in
  let pi=≪ $pid$ mod sqrt_p≫
 and pj=≪ $pid$/sqrt_p≫  in
   let fromA=≪ fun l→ (($pi$+$pj$+l) mod sqrt_p)*sqrt_p+$pi$≫
  and fromB=≪ fun l→ (($pi$+$pj$+l) mod sqrt_p)+$pj$*sqrt_p≫
 in
  for l=0 to sqrt_p−1 do
   let rcvpA,rcvpB = super (fun ()→  get_from fromA parA l)
                           (fun ()→  get_from fromB parB l)
    in ≪ $parC$ := seq_plus $parC$ (seq_mult $rcvpA$ $rcvpB$) ≫
  done;
  parC
```

```
void c_mat_mul(double *A,
               double *B,
               double *C,
               int n,int p_sqrt)
{
 register int pid;
 double *a,*b,*c;
 register int l,pi,pj,ni,size
 ni=n/p_sqrt;
 size=ni*ni*sizeof(double);
 pi=pid % p_sqrt;
 pj=pid / p_sqrt;
 a = (double *)malloc(size);
 b = (double *)malloc(size);
 c = (double *)malloc(size);
 init_mat_float(C,ni)
 bsp_push(A,size);
 bsp_push(B,size);
 bsp_sync();
 for (l=0;l<p_sqrt;++l)
  {
bsp_get((((pi+pj+l)%p_sqrt)*p_sqrt+pi),A,0,a,size);
bsp_get((((pi+pj+l)%p_sqrt)+pj*p_sqrt),B,0,b,size);
   bsp_sync();
   mat_seq_mult_float(a,b,c)
   mat_seq_add_float(C,c)
  }
 bsp_pop(B);
 bsp_pop(A);
 free((void *)b);
 free((void *)a);
 free((void *)c);
}
```

Figure 2.7. BSML code of the matrix multiplication (left) and in C +BSPLIB (right).

implemented with a BSML parallel vector of normed field element arrays. Linear algebra algorithms are functors which perform computation on matrices made of normed field elements.

Our example is based on an algorithm presented in [93] which is independent of type of data of the square matrices. The true algorithm implemented in our library is close to it except that it works for arbitrary size matrices and used a scattered distribution and not a block one — it is thus less readable.

In Figure 2.7, we give the code of this algorithm using BSML and C +BSPLIB [25]. One can remark that each processor receives data from two distinct processors at each super-step due to a *round-robin* distribution of the blocks. For the BSML implementation, we use twice the "get_from" utility function and we "superposed" them to avoid the duplication of super-steps — see in Section 2.2.2 for a definition of the **super** primitive. Each "get_from" is built as in the algorithm and as the **bsp_get** in the C code. In this C code, we used specific sequential addition and multiplication of matrices of floats. These two codes have two main differences. First, using our modular implementation, the BSML code abstracts the type of the elements of the matrices. Second, in the C code we modify in place the matrix while in the BSML code, the resulting matrix is build dynamically.

Figure 2.8 gives some benchmarks for matrices of floats (times and speedups). Note that if we want other kinds of matrices, using the modular BSML code, we just have to change the instantiation of the algebra while using C code, we have to change the code itself. We remark that the C +BSPLIB code is three timefaster than BSML one. This mainly due to the use of functors that are known by the OCAML community to slow down the programs. For both, speedup are close to the linear acceleration which is not surprising since matrix multiplication is a good parallel problem. We can also see a super-linear acceleration for the BSML code which is more scalable than the C code. This is mainly due to the fact that the computational parts of the C code are efficient enough (even in the sequential implementation) to be only slow down by the communications.

We have also implemented a LU decomposition, a Cholesky factorisation, a Gauss/Jordan elimination, transposition and inversion of matrices. Many other operations can be easily implemented from this set of operations. For sack of concision, we do not present them. They are described in [C7].

2.1.6 Past Syntax

From Publications 3

This section summarises [W3].

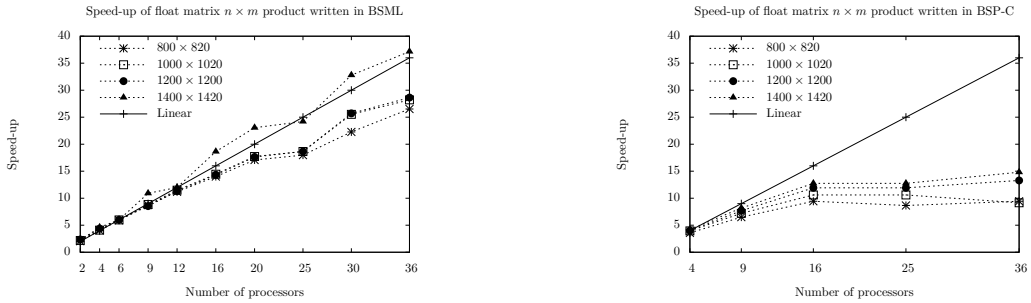


Figure 2.8. Multiplications of Dense Matrices of Floats.

In [80], BSML had the following primitives:

mkpar: (int → 'a) → 'a par
apply: ('a → 'b) par → 'a par → 'b par
put: (int ∗ 'a list) par → (int ∗ 'a list) par
at: bool par → bool

where **mkpar** create a parallel vector, **apply** allows local application of a vector of functions to a vector of values, **put** performs the exchange of data (each processor contains a list of associations "to processor id" of sending values and the return is a list of received values) and **at** allows extract a boolean and must be used in a "if" statement. Since [M1], the primitives of communications are as presented before.

However, the use of these past primitives can sometimes become awkward. Indeed, every operation inside of parallel vectors has to call a primitive and define an "ad hoc" function. This gets worse when working with multiple vectors, with nested calls to **apply**. Simply transforming a pair of vectors into a vector of pairs is written:

```
(* parfun: ('a → 'b) → 'a par → 'b par *)
let parfun f v = apply (mkpar (fun _ → f)) v
(* parfun2: ('a → 'b → 'c) → 'a par → 'b par → 'c par *)
let parfun2 f x y = apply (parfun f x) y

let combine_vectors(v, w) = parfun2(fun v w → v, w) v w
```

which is unsatisfactory because we have to define, each time, a specific function. This implies creating named parameters although our function will only be applied to our vectors, and can be confusing:

```
let combine_vectors(v, w) = parfun2 (fun w v → w, v) v w
```

which is the "same" as above but can lead the programmer to errors. Novice BSML programmers offer find these two aspects difficult to deal with.

So, instead of a point of view based on primitives, we can consider the execution levels such that one can declare code that will be executed globally as in standard OCAML and code that will be executed locally, inside a parallel vector. Then, to access to local data in a local section, we need no more to define additional functions because opening vectors now can be done locally. This why we finally introduce the explicit creation of vector $\ll$ e$\gg$. We will see that it also deals to simplify some extensions of BSML notably parallel exceptions. In practise, we believe simple syntax may be as important as simple semantic.

2.2 Extensions

2.2.1 External Memory Operations and Imperative Features

From Publications 4

This section summarises the works of:

- *[C18] for the interaction of imperative features with* BSML*;*
- *[C16] and [R4] for external memory operations in* BSML*.*

(a) Imperative Features

OCAML provides several very useful imperative features. The most commonly used are data structures that can be modified in-place (arrays, strings, mutable record fields and references). To make things simpler, we only consider references, as the others can be easily reduced to them.

References are manipulated through three operators: **ref**, to create a reference, := to change the value stored in one, and ! to get the value currently stored.

In [C18] and [73], we have show (and give a formal semantics) that the adding of imperative features in BSML implies, as might be expected, two levels of memory: local and replicated. Both kind of references must be manipulated in their own contexts and replicated references could also be read in the local context. Thanks to the possibility of serialisation of OCAML values, references can be send: a copy of the reference is done and could be thus manipulated by a remote processor. But, as we will show in Section 2.3, references can easily cause a loss of replicated consistency.

(b) External Memory in bsml

Many important computational applications involve solving problems with very large data sets [94]. Such applications are also referred as *out-of-core* applications and can involve data sets that are too large to fit in the main memory. To increase the available memory size, the algorithms must be restructured with explicit I/O calls on this secondary memory that is mainly disks. Parallel disk I/O has been identified as a *critical component* of a suitable *high performance* computer. [95] showed how an EM (external memory) machine can take full advantage of parallel disk I/O and multiple processors. This model is based on an extension of the BSP model for I/O accesses. Our research aims at combining the BSP model with functional programming. We naturally need to also extend BSML with I/O accesses for programming EM algorithms.

In [R4], we have show that two kinds of I/O operations are needed for a replicated consisting and thus a determinist semantics. The cost model of these new primitives and a formal semantic have been investigated. In addition, some benchmarks have been done. Such primitives need occur in their own context (local or global). Local (asynchronous) I/O operators work on local disks whereas global I/O operators work on share disks and are synchronous operations — except for reading and writing values. This is common in parallel programming where the user's data are stored in share disks.

2.2.2 Superposition of Parallel Computations

From Publications 5

This section mixes the following publications:

- *in [C12] for the first implementation of the superposition;*
- *in [R1] and [W5] for a new implementation which used a* CPS *transformation.*

(a) A Primitive of Parallel Composition

BSP paradigm's simplicity and elegance comes at a cost: the ability to synchronise a subset of the processors breaks the cost model — even if this feature can be useful for iterative programming, see next chapter and discussion in the conclusion. *Subset synchronisation* is generally used to recursively decompose computations into independent tasks — the divide-and-conquer paradigm. In [96], the author proposes a natural way to fit divide-and-conquer algorithms into the BSP framework without subset synchronisation by using sequentially interleaved threads of BSP computation, called *super-threads*. The last BSML primitive called **super** allows the evaluation of two BSML expressions E_1 and E_2 as super-threads [90] [M1]. From the programmer's point of view, the semantics of the superposition is the same as pairing *i.e.*, building the pair (E_1, E_2) but of course the evaluation of **super** E_1 E_2 is different — see Figure 2.9.

In the left, pairing is just the sequential evaluation of the two programs. In the right, using the superposition, the phases of asynchronous computation of E_1 and E_2 are run. Then the communication phase of E_1 is merged with that of E_2. The messages are obtained by simple concatenation of the messages of each super-thread and only one barrier occurs. If the evaluation of E_1 needs more super-steps than that of E_2 then the evaluation of E_1 continues — and *vice versa*. The parallel superposition is thus less costly than the evaluation of E_1 followed by E_2.

As example, the parallel prefix computation can be done using a divide-and-conquer BSP algorithm (with **super**) where the processors are divided into two parts and the scan is recursively applied to those

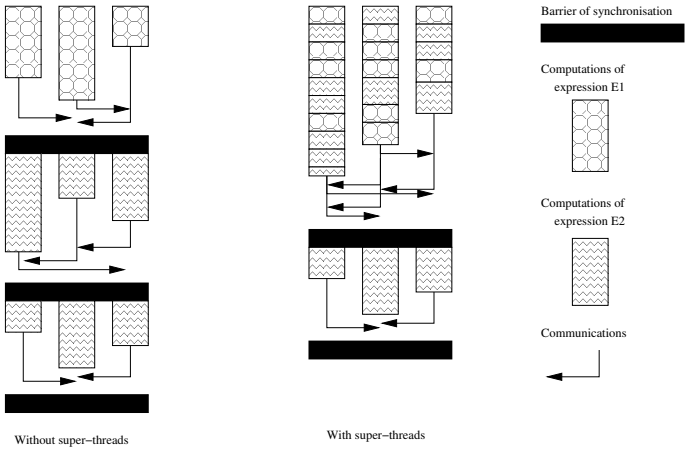


Figure 2.9. The evaluation of the superposition for two super-threads on three processors.

parts; the value held by the last processor of the first part is broadcast to all the processors of the second part, then this value and the values held locally are combined together by the associative operator $\oplus$ on the second part. Figure 2.3 (right) gives the code of this method.

(b) Older Superposition Implementation

In ML-like languages, it is straightforward to add imperative features. Using concurrent threads can thus introduce non-deterministic results — deadlocks and data-race. To avoid this, a strategy for the choice of the unique active super-thread has been added [M1]: the active super-thread is evaluated until it ends its computations or it needs communications. When communications are done, the first super-thread which has finished "its past super-step" is re-evaluated, *i.e.*, it becomes the new current active super-thread.

Currently, based on a semantics study, the superposition is implemented using system threads [C12]. Each time a superposition is called, a new thread is created and share locks are used each time a communication primitive is called. Some benchmarks have been done on the prefix computation and this implementation of the superposition were used in [R3] for balancing dynamically parallel data structures.

There is an important drawback to this method. System threads slow down the running of an OCAML program: a global lock is used due to the GC of OCAML. When using many threads (*e.g.* > 100), the performance of the program totally collapses. This greatly limits the usefulness of the superposition when a large number of super-threads are run simultaneously. We now present another implementation which uses a global continuation-passing-style (CPS) transformation on the original code.

(c) New Implementation of the Superposition

In the following, we use only two parallel primitives: **yield** and **super**. Here, **yield** replaces communication primitives, abstracting away communication handling. **yield** suspends the currently executing super-thread (called thread in the next) and schedules the execution of the next thread, as defined by the semantics of **super**. We also only present the transformation (new implementation of the superposition) of the code for a core language — see [R1] for more details.

Continuation Passing Style. The original CPS transform [97] was designed to study the various evaluation strategies for the lambda-calculus by making the control explicit, as a *continuation*: a function representing the evaluation context. It was then discovered that giving to the programmer or the compiler writer the ability to explicitly manipulate continuations was an expressive tool to perform various analysis or to encode various high-level constructs, such as exceptions or light-weight threads [98]. Below is the original CPS transformation, as defined in [97] for λ-calculus:

$$\begin{array}{lcl} [\![x]\!] & = & \lambda k.(k\ x) \\ [\![\lambda x.M]\!] & = & \lambda k.(k\ (\lambda x.[\![M]\!])) \\ [\![(M\ N)]\!] & = & \lambda k.([\![M]\!]\ (\lambda m.([\![N]\!]\ (\lambda n.(m\ (n\ k)))))) \end{array}$$

```
yield  =  λk.Waiting k
super  =  let loop = fix loop λr1.λr2.bind yield (λ () .match (r1, r2) with
                    | (Terminated x1, Terminated x2) → ret (x1, x2)
                    | (Terminated _, Waiting s) → loop r1 (s ())
                    | (Waiting s, Terminated _) → loop (s ()) r2
                    | (Waiting s1, Waiting s2) → loop (s1 ()) (s2 ())) in
                                ret λf1.ret λf2.
                                  let r1 = ((ret f1) @ (ret ())) (λx.Terminated x) in
                                  let r2 = ((ret f2) @ (ret ())) (λx.Terminated x) in
                                    loop r1 r2
```

Figure 2.10. Superposition using the monadic/CPS implementation.

Use of monadic operators. Our CPS transformation for implementing the superposition uses modads. Monads allow to extend a language while enforcing a correct operational behaviour [99]. A monad is the data of three primitives: **run**, **ret** and **bind**, operating on a type $M\,\alpha$. **run** has type $\forall\alpha.M\,\alpha{\to}\alpha$ and executes a monadic program. **ret**, of type $\forall\alpha.\alpha{\to}M\,\alpha$ transforms a base value into a monadic one. Finally, **bind** allows chaining monadic computation as reflected by its type $\forall\alpha,\beta.M\,\alpha{\to}(\alpha{\to}M\,\beta){\to}M\,\beta$.

Our threads are modelled as resumptions, meaning that they are in a suspended state or terminated: **type** 'a thread=Terminated **of** 'a | Waiting **of** (unit→ 'a thread) The monadic type is always *thread*: $M\,\alpha = \forall\,\beta.(\alpha \to thread\,\beta) \to thread\,\beta$. The monadic primitives are thus defined as follow:

```
ret x     =  λk.kx
bind m f  =  λk.m(λv.fvk)
run       =  λx.((fix loop λt. match t with
                              | Terminated x → x
                              | Waiting s → loop (s ())) (x (λx.Terminated x)))
```

We now straightforwardly proceed to the definition of the monadic transformation on expressions $T_0[\![e]\!]$, fully defined in [R1]. The primitives **yield** and **super** (in Figure 2.10) are then defined using first class continuations where $a\,@\,b \equiv$ **bind** $a\ (\lambda v_a.$**bind** $b\ (\lambda v_b.(v_a\,v_b)))$.

The operational behaviour of these primitives is clear: **yield** captures its own continuation, and stores it into a suspension for further evaluation; **super** first suspends its own execution (using **yield**), then schedules the execution of its two sub-threads until they are terminated. We have the following correction result for our core-language:

Theorem 1

If $e \Rightarrow v$ *then* $T_0[\![e]\!] \approx$ ***ret*** v.

where $\Rightarrow$ is a small step semantics and $\approx$ an equivalence of expressions — see [R1] for more details.

Optimisation. But our wanted implementation aims to preserve as much code as possible. To illustrate the problem of this naive transformation, we give this trivial example:

$$\begin{array}{lcl} \textbf{apply} & = & \lambda f.\lambda x.fx \\ T_0[\![apply]\!] & = & \lambda k_0.k_0\lambda f.\lambda k_1.k_1\lambda x.(\lambda k_2.(\lambda k_3.k_3 f)(\lambda v.(\lambda v_f.\lambda k_4.(\lambda k_5.k_5 x)(\lambda v.(\lambda v_x.v_f\,v_x)vk_4))vk_2)) \end{array}$$

It is obvious on this code that this transformation can not be used as is: a code that not contains any concurrency primitive need not to be converted. The full transformation of a program to CPS considerably impedes performance. This overhead is usually alleviated using transformation-time reductions to eliminate these so-called "administrative redexes" on the programs. Observing how some very limited parts of the program need continuations, it seems natural to try to convert only the required expressions — in our case, only **yield** and **super** need them. We thus need a partial CPS transformation [100]. The expressions to be transformed are those susceptible to reduce a **yield** or **super** expression.

Since we must cope with higher-order functions, the partial CPS transformation is guided by a flow analysis which yields a straightforward flow inference algorithm whose purpose is to decide if an expression is susceptible to reduce a **yield**: we tag it as *impure* — *pure* otherwise. Our type system is derived from the type system for CFA defined in [101] and the rules and implementation details can be found in [R1]. Mainly we perform a (global) monomorphisation of the code and a code duplication where a high order function is used in a impure context or in a pure one — use the superposition or not. Take for example, the previous code. It could be use in a context without a use of the superposition. It still unmodified. Else, we have this variation:

$$\begin{array}{lcl} apply_{\mathcal{I}} & = & \lambda f.\lambda x.\texttt{let}\ () = \textbf{yield}\ \texttt{in}\ fx \\ T_1[\![apply_{\mathcal{I}}]\!] & = & \lambda k_0.k_0\lambda f.\lambda k_1.k_1\lambda x.\lambda k_2.(\lambda k_3.Waiting\,k_3)(\lambda().(\lambda().\lambda k_4.k_4(f\,x))\,()\,k_2) \end{array}$$

There is still many administrative redexes, but pure expressions are preserved from being transformed. On real-world programs, most of the computation takes place in pure expressions, making the CPS part less of a burden. We thus defined an optimising CPS transform which creates no administrative redex (adapted from [102, 103]) and which is equivalent to the initial one. For sack of brievty, we do not give the details. Currently our implementation works on a large subset of OCAML without objects, labels and functors.

For programs that use a small number of super-threads, this new implementation of the superposition does not improve the performances. This is mainly due to the fact that most of the computations are done within parallel vectors: replicated code is most of the time used to coordinate the works of the processors. Thus expressions within vectors are pure and are not transformed: our type inference allows that. The performances should clearly improve whenever the number of super-threads is great. This will be show latter for the implementation of data-flow skeletons.

2.2.3 Parallel Pattern-matching

We now present how the explicit use of parallel vectors allows an interesting extension: a notion of parallel pattern matching for BSML.

Parallel pattern-matching. Standard matching of OCAML allows the programmer to take a decision over a local or a replicated value. But this is not sufficient to take a decision like "if one processor have a specific value and other not" or "all processors have a specific value". For this, we need to match parallel vectors. In [C20], we have adding matching like ≪ pattern_1≫ | ≪ pattern_2≫ | ... that is each pattern is a test for matching a value everywhere — on each processor. This is clearly not sufficient and too less expressive for a gentle programming.

The main idea is to search for values corresponding to a ground in a vector without using the order of the elements of the vector. It is introduced with a construction similar to the **match with** of OCAML:

```
matchpar e withpar
 | parallel_pattern1 → e1
 | parallel_pattern2 → e2
 ...
```

A parallel pattern, among ≪ ≫ is a list of patterns each separates by a \. These patterns will be searched successively in the vector. They can be either a standard OCAML pattern for testing if the pattern matchs on a processor; or a pattern with a star (pattern)∗ for any number (strictly positive) occurrences of the matching in the vector; or the default pattern _ for any content.

For examples, the pattern ≪ (None)∗≫ matchs a vector whose elements are all equal to None, and ≪ None \ _ ≫ matchs a vector with at least one None.

As patterns in OCAML, a parallel pattern can bind variables. For example ≪ Some x \ _≫ binds the variable x to the first instance of the constructor Some found in the vector. The result is a bit more complex for repeated patterns. For example ≪ (Some x)∗ \ _ ≫ should bind the variable x to different unknown (*a priori*) values. The behaviour of this filtering is allocating a list for all variables appearing in a star: x is here the list of the corresponding values.

The filter works by elimination. In case of overlapping, only the first accepting pattern in the list is considered. Thus, the pattern ≪ (0)∗ \ (x)∗ ≫ bind x to the list of non-zero values of the vector since any zero will be eliminated by the first pattern. Finally, it is also sometimes useful to know which processor match a pattern. The syntax is extending using the construction **at** p, where p is a variable which bind the pids of the processors that match the pattern. It has the advantage to store informations about the processors involved. We now give some examples. Ones can now code a total exchange as:

```
(* total: 'a par → 'a list *)
let total vl = matchpar vl withpar ≪ (x) ≫ → x
```

In a similar way, ones can test if all elements of a vector satisfies a condition:

```
matchpar ≪ is_empty $v$ ≫ withpar
  | ≪ (true)∗ ≫ → e1
  | ≪ (false at p)∗ \\ _ ≫ → e2
```

Without parallel pattern matching, it requires the definition of complex ad-hoc operators:

```
let new_is_empty pid v = if (is_empty v) then [] else [pid]

let op e1 e2 = match e1,e2 with
 | [],[] → []
 | hd::tl,[] → hd::tl
 | [],hd::tl → hd::tl
 | _ → List.concat e1 e2
```

```
let my_list = simple_reduce op (apply (mkpar (fun pid → is_empty pid)) v) in
 if (List.empty my_list) then e1 else e2
```

As last example, the OCAML's "List.assoc" but for a distributed list can be coded:

```
let assoc_par x vl =
  let vassoc = ≪ try Some (List.assoc x $vl$) with Not_found → None ≫
  in matchpar vassoc withpar
     | ≪ Some x \\ _ ≫ → x
     | ≪ (None)* ≫ → raise Not_found
```

Parallel matching is only a syntactic sugar since it does not add any functionality related to parallelism. But it greatly simply the writing. The implementation (which uses CamlP4) is describe in [73].

2.2.4 Parallel Exception Handling

From Publications 6

This section summarises [R2] and [C13] and describes the design and the implementation of parallel exceptions in BSML.

(a) Problematic

Exceptional situations (in a in-depth search for example) and errors (*e.g.* a Stack_overflow) are handled in OCAML with a powerful system of exceptions. In parallel, this is at least as relevant: if one processor triggers an exception during the course of a parallel computation, BSML has to deal with it, like OCAML would, and prevent a crash. Take for example the following code:

```
let vs = ≪ if $pid$=0 then (raise Failure) else (fun _→ $pid$)≫
 in put vs
```

Here, an exception is raised locally on processor 0 but other processors continue to follow the main execution stream, until they are stopped by the need of synchronisation. Then, a deadlock occurs. If the same code had been enclosed in a **try** ... **with**, processor 0 and other processors would have branched into different global execution streams, the normal one and the exceptional one, leading to a replicated inconsistency: they could have a different number of super-steps which is not possible in the BSP model.

(b) An Exception Mechanism for BSML

The solution we provide intends to stay as familiar as possible to the programmer: the missing piece to a parallel exception system is a way to catch globally exceptions that are raised locally. The catching of local exceptions is as follow:

```
trypar
  ≪ if $pid$=0 then (raise Failure) else 1≫
withpar
 ≪ Failure ≫ → print_string "A␣proc␣fails"
```

The structure **trypar**...**withpar** is used to safely recover this local exception, globally. Replicated local exceptions caught this way depends of the parallel patterns. Note that in [R2], catch exceptions could only be used as a set. Using our parallel patterns, exceptions are handly as pattern matching but the underlying implementation continues to used sets.

We will consider two points carefully: (a) a local exception should never prevent replicated code from being executed, or the system becomes inconsistent; (b) at the end of the super-step, a local exception has to be treated by all processors.

Since replicated and local code may be juxtaposed in the same super-step, we need to get aside from the standard exception handling techniques to ensure that replicated code is run normally even after a local exception. During a super-step, there might be local and replicated exceptions coexisting and they must be treated at different levels: a replicated exception, since it is raised by all processors, is treated immediately in the OCAML way. A local exception, on the other hand, must not hinder the global behaviour of the processor yet, so it is kept silent to replicated code until the end of the super-step. This means, in particular, that a processor in a state of exception may not perform any local computation until the next synchronisation.

At the end of a super-step, the exception state is communicated to all processors to allow a global decision to be taken. In such a situation, the local results obtained are partial, inconsistent or nonexistent. Although we are discussing a way to enable the program to recover them afterwards, we currently adopt the standard approach and discard them, switching to the exceptional treatment specified by the user.

Local exceptions are thus *deferred* until the end of the super-step. However, it is undesirable that an exception escapes the scope of the **trypar**...**withpar** it was raised in. For this reason, communications (and a barrier) must be forced at the **withpar**. In the case where the superposition is used, some super-thread could also be in different states. We have also make the choice to prioritise the exceptional treatment specified by the user. The formal semantics and the implementation are described in [73].

To summarise, the implementation is a combination of an added protection on the primitives, an extended communication protocol that can be triggered to global exchange of exceptions, and various rewriting rules that both add new constructs and ensure the non-interference of this mechanism with the user program. The implementation used CamlP4 to transform the used of parallel exceptions into a standard use of exceptions with appropriate communications. Our example is the parallel version of the "List.assoc" which can now be code as follow (assuming a defined exception "Found"):

```
let assoc_par x vl =
  trypar ≪ raise (Found (List.assoc x $vl$)) ≫ withpar
    | ≪ Found y \\ _ ≫ → y
    | ≪ (Not_found)* ≫ → raise Not_found
```

The local expression raises the exception "Not_found" or "Found" if a result was found. The parallel patterns allow to quickly distinguish between these two cases, "a result was found" or "no processor has found a result".

(c) An Example of the Use of the Exception Handling

We illustrate the used of parallel exceptions in BSML by the implementation of sudoku problem using a generic parallel backtracking algorithm — for brute force search.

A naive solution. Backtracking consists in searching for a solution by exploring a tree of possibilities depth-first. This is close to the parallel model-checking problem of Chapter 4 and it is well known that it is not efficiently parallelizable [104]. For this example, assume the problem provided in the form of the root of a tree and a child function that tests if a node is a solution (and raise an exception if it is) or returns his sons in the form of a list. A sequential solution could be coded as follow:

```
let rec down n = ignore (List.map down (childs n))
let backtrack () = try down root; None with | Solution_found s → Some s
```

where the down is responsible of the recursive calls for the exploration of the tree. It returns no result. It is the exception raise by childs that deal with it when necessary. The naive parallel version follows the same process, except that the nodes are distributed over a parallel vector. But the parallelisation of this process explores the children of several different nodes at the same time:

```
let backtrack () = trypar
                     let init = select_list (childs root) in ≪ down $init$ ≫ ; None
                   withpar ≪ Solution_found s \\ _ ≫ → Some s
```

When one of the processor found a solution, it locally throw the exception Solution_found to be catch.

This simplistic approach has two main defects. First, it does not balancing, and therefore assumes that all initial nodes can be distributed fairly to all processors and the local sub-sets will result in a comparable amount of calculations. Second it is fully asynchronous, and works in a single super-step: following our model of exceptions, a solution can be propagated at the end of this super-step and therefore once all processors have finished their calculations. The duration will be the same if a processor finds a solution quickly, and almost all of the tree will be explored in all cases. The challenge is to fairly allocate the nodes to be explored between the different processors and rebalanced when necessary.

A more efficient parallel version. A more advanced solution is to split the computation in super-steps, making them a part of exploration followed by an eventual re-balancing. Each re-balancing will be an opportunity to detect a possible solution. The cutting works as follow: it keeps the local calculations at the same level of depth in the tree. The rate of connection, although aleatory in general, depends on most of the depth. By exploring these levels one after another, we reduce the chances of a quick gap between the load of the processors, and thus limit the need for re-balancing. At each super-step, the parallel machine will explore a forest corresponding to sub-trees whose roots are subsets of nodes at level n in the tree to explore. This super-step ends when all the descendants of level n of these nodes have been calculated, which are rebalanced and partitioned, and giving rise to a new super-step.

Figure 2.11 gives the generic code. The algorithm uses two mutually recursive functions. The first one, "down" is relative to the depth of the tree and starts to proceed by re-balancing the nodes. The second

```
let rec down nodes = (* depth−recursive *)
  matchpar ≪ List.length $nodes$ ≫ withpar
    | ≪ (0)* ≫ → () (* backtracking *)
    | ≪ (nl)* ≫ → let nodes, ntot, max_sz = rebalance_if_needed nodes nl
                  in explore nodes max_sz
and explore nodes max_sz = (* width−recursive *)
  let split_nodes = ≪ rev_split [] nchld $nodes$ ≫ in
  let current_nodes, next_nodes = ≪ fst $split_nodes$ ≫ , ≪ snd $split_nodes$ ≫
   in down ≪ childs $current_nodes$ ≫ ;
      if max_sz − nchld <= 0 then ()
       else explore next_nodes (max_sz − nchld)

let backtrack () =
  trypar
    let init = ≪ if $pid$=0 then childs root else [] ≫ in down init; None
  withpar
    | ≪ Solution_found s \\ _ ≫ → Some s
```

Figure 2.11. Backtracking using parallel patterns.

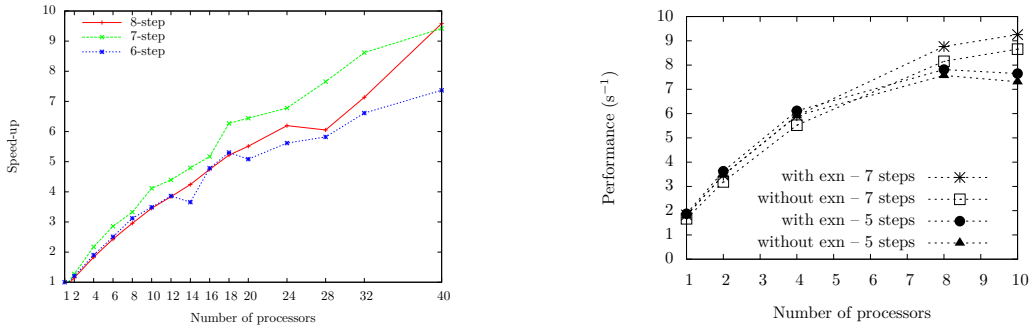


Figure 2.12. Solving a 16×16 sudoku (left) and 9×9 but using or not parallel exceptions (right).

one, "explore" is responsible for the exploration in breadth: it recursively selects some of the nodes on each processor using split_nodes and carry on an in-depth exploration using "down".

Each function returns unit when no more nodes need to be explored, leaving the caller to resume the exploration. The reader may be surprised by the fact that we impose a barrier of synchronisation at each level of descent in the tree. When applying the algorithm to a given problem, we compose several times the childs function so to get a connection rate sufficient for balancing. The frequency of barriers can be adjusted by the "nchild" parameter.

It remains to define a function able to re-balance the data. In our case, the list is not ordered, we thus calculate the number of data on each processor, and move the excess of data of a processor i to the first processor "cyclically to the left" of i which has a deficiency number of data.

The communications are then performed and re-assembled as a distributed list. It remains to determine a criterion of balance, to avoid unnecessary communication. Our simple criterion produces a balanced, while the differences between the amounts of local data on the total load is too large, but by allowing at least a gap of 1 for small sets.

Results. As an example, we used this generic backtracking algorithm to solve sudoku with brute-force. Sudoku is a fashionable game that consists in filling a $n^2 \times n^2$ grid with integers from 1 to n^2 according to constraints that ensure, given some initial numbers, that only one solution is possible. The children are generated by trying every number at each free square and checking for validity. A mild optimisation consists in composing the children function several times to obtain enough nodes for an even distribution between processors, which becomes mandatory when increasing the size of the machine.

Figure 2.12 (left) shows the results for the cluster LACL2 using MPICH. The execution time of just searching a solution is too uncertain to have any real meaning in terms of performance – especially as the $n-1$ nodes are explored in a different order depending on the number of processors. We have therefore chosen to modify the program to count the number of nodes explored per second, the acceleration is in relation to this number — which exceed 500 000 for 40 cores. The accelerations do not scale very well. This is mainly due to a naive balancing algorithm: too many node are exchanged at each barrier. Better algorithm could be find for specific NP-problems using realistic heuristics (and not a generic brutal force method) but this is not the subject of this section.

Another version of our naive solver, without any use of exceptions, was implemented. The main descending function had to gather the results of all processors and check if there was a solution at one of them. Accordingly, the size of the backtrack function was increased from 26 lines of code to 44 – exceptions made us save 40% in code size on this example. We solved a given grid of dimensions 9×9 using the MPICH library on the cluster LACL1 and using the two implementations — with or without the use of parallel exceptions. Figure 2.12 (right) shows the performance in seconds^{-1} depending on the number of processors (so that a linear speedup would be a straight line), for two different levels (steps) of the latter optimisation. This results are the median of a large number of experiments. We notice little impact on performance between the versions with and without exceptions, which is sound since the algorithm is not changed; better, the difference is very stable and in favor of the version with exceptions: we explain it by the added checks that have to be made to extract the possible results at every step of computation.

2.3 Type System

From Publications 7

This section summarises [W1].

BSML is currently implemented as a library for OCAML and given the extensions that have been added to BSML recently (new primitives, I/O, references, exceptions, superposition, *etc.*), BSML does not have a type system that is able to prevent run-time problems due to an incorrect use of the parallelism in standard user programs — it currently only uses the one of OCAML. It is thus currently the responsibility of the programmer to avoid these problems. A previous static analysis [R6] achieved this goal in a smaller language by building constraints along expressions, but it is not relevant with the current version of BSML anymore; the system presented here is less restrictive and more scalable, and is inherently compatible with the new features. We do not give all the details of the type system. It is fully available in [W1] and in [73].

2.3.1 Different Kinds of Unsafe BSML Programs

Programs in BSML are supposed to follow some invariants in order to keep the properties of BSP and the strict distinction between local and replicate executions. We first describe these invariants, and how they can be broken by programs that would be correctly typed in ML.

(a) Execution of Local Primitives Outside Parallel Vectors

As it may already appear to the reader, the most important one is the consistency of replicated expressions. For example, in the following program (where v is a parallel vector) "**if** a the **put** v **else** v", an inconsistent value for a would cause a different branch of execution at different processors, some of them waiting at a barrier (when $a = true$) and the others not, causing a deadlock. Loss of replication happens for example when using non-deterministic operations, *e.g.* random, or operations that are dependant on the local processor, *e.g.* system or I/O related.

(b) Local Execution of Parallel Primitives

proj (or **put**) should not be evaluated in the context of a vector. For example, if vec' as type t par for a given type t, the following code is incorrect: **let** vec = $\ll$ **proj** vec' 0$\gg$ but should be written: **let** vec = **let** x = **proj** vec' 0 **in mkpar**(**fun** i $\rightarrow$ x) since the evaluation of an application of **proj** requires communication and synchronisation, if only one processor execute the proj, a deadlock occurs.

BSML is built around the notion of global execution and parallel vectors. This makes the semantics of programs such as $\ll$ (**fun** i $\rightarrow$ $\ll$ **fun** j $\rightarrow$ i$*$j $\gg$) **pid**$\gg$ ambiguous: the parameter of the function of the first vector is supposed to be executed locally. In the second vector,the code would thus be executed once on each processor; since this is a new parallel vector, that would require each of them to communicate with its neighbours to create locally a new global object, actually creating **p** parallel vectors containing **p** values each. The semantics of such a behaviour would be very complex, not to mention the loss in readability, performance and performance prediction possibilities.

The local execution of the other primitives would be even more difficult to make sense of: what would be the meaning of **put** used locally by a processor i and requesting communications between processors j and k? To keep the language and the semantics simple and usable, the type system rules out local execution of the primitives, which will be called *invalid parallelism*. Note that the NESL language allows

(and encourages) this kind of nested parallelism. But the cost model is more complicated since it is impossible to known if an expression would perform communications or not.

(c) Nested Parallel Vectors

Preventing invalid parallelism still allows the definition of nested parallel vectors, because global values can be accessed anywhere from the local code nested evaluation of an expression of type (t par) par for a given type t would lead to an unspecified behaviour. For example, the code **let** v = ≪ $**pid**$≫ **in** ≪ v ≫ is a valid program that would supposedly return the value $w = \langle\langle 0, \ldots, p-1\rangle, \ldots, \langle 0, \ldots, p-1\rangle\rangle$. This, however, does not faithfully describe the actual data since by definition each processor only holds one value of a vector. The data stored would rather look like $\langle\langle 0\rangle, \ldots, \langle p-1\rangle\rangle$. The expression **proj** w instead of returning the expected vector v, thus returns the result $\langle 2, \ldots, 2\rangle$, which is a parallel vector which has actually lost its parallel information.

2.3.2 Definition of the Type System

(a) Inference Rules

We use a Hindley/Milner type system inspired from that of ML, with added effects [105, 106] and constraints [107]. In this section, we will first concentrate on the prevention of invalid parallelism, then introduce more constructs that are needed to avoid nested parallelism. In the core language described above, types are defined by:

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \textit{type variable} \\
 & | & \mathsf{Base} & \textit{base type} \\
 & | & \tau \xrightarrow{\pi} \tau & \textit{function} \\
 & | & \tau \times \tau & \textit{pair type} \\
 & | & \tau\ \mathsf{par} & \textit{parallel vector}
\end{array}
\qquad
\begin{array}{rcll}
\Lambda & ::= & \delta & \textit{locality variable} \\
 & | & \ell & \textit{local} \\
 & | & g & \textit{global}
\end{array}
$$

The first and main concern of our type system is to prevent local execution of parallel primitives. For that purpose, we use effects. We write $\Gamma \vdash e : \tau/\pi$ if the expression e is typed τ with effect π in the context Γ. An effect π can remain latent in a function, in which case we write $\tau_1 \xrightarrow{\pi} \tau_2$.

We define *locality* Λ as given above: locality variable δ behaves in a way similar to type variables and takes values in Λ. Though they will be extended later, for now we consider simple effects indicating only the current locality, and take $\pi = \Lambda$.

We classically used the common notions of *type scheme* (σ), *typing context* (Γ), *substitution* (φ) and *free variables* $\mathcal{L}(\tau)$ which are defined in the usual way. We define an *instance* as: given a type scheme $\sigma = \forall\alpha_1 \cdots \alpha_k\delta_1 \cdots \delta_l.\tau$, we say that τ' is an instance of σ and write $\sigma \leq \tau'$ if, and only if there exists a substitution φ on $\{\alpha_1 \cdots \alpha_k\delta_1 \cdots \delta_l\}$ (by which we mean that any variables not in this set are left unchanged by φ), such that $\tau' = \varphi(\tau)$.

The locality of an expression denotes its presence on a single processor or on all of them. A function does not have a specific locality as a value (it is usually defined globally), but may be restricted to a specific one for its execution. Since this cannot be seen from its return type alone, we use latent effects on function arrows to denote it. Some functions (like the identity and most sequential functions) do not have a specific locality, in which case their locality is a free variable.

Other functions, on the other hand, may be inherently local if they cannot consistently be applied in a replicated way on all processors. The function random, for example, would not return a valid replicated value if executed by all processors: we give it the type $unit \xrightarrow{\ell} int$. Operators need to have statically defined types. These are stored in a map TC, from operators to type schemes:

$$
\begin{aligned}
TC(\texttt{put}) &= \forall\alpha\delta.(int \xrightarrow{\ell} \alpha)\ \mathsf{par} \xrightarrow{g} (int \xrightarrow{\delta} \alpha)\ \mathsf{par} \\
TC(\texttt{proj}) &= \forall\alpha\delta.\alpha\ \mathsf{par} \xrightarrow{g} int \xrightarrow{\delta} \alpha \\
TC(\texttt{super}) &= \forall\alpha\beta\delta_1\delta_2.(unit \xrightarrow{\delta_1} \alpha) \xrightarrow{g} (unit \xrightarrow{\delta_2} \beta) \xrightarrow{g} \alpha \times \beta) \\
TC(\texttt{fst}) &= \forall\alpha\beta\delta.\alpha \times \beta \xrightarrow{\delta} \alpha
\end{aligned}
$$

These types make the locality constraints of the primitives clear: they are all restricted to global execution, while the functions they take as parameters must accept to be executed in a local scope, *i.e.* must not themselves execute primitives.

$$\text{ENVTYPE}\ \frac{\Gamma(x) \leq \tau}{\Gamma \vdash x : \tau/\pi} \qquad \text{LET-IN}\ \frac{\Gamma \vdash e_1 : \tau_1/\pi \qquad \Gamma; (x : Gen(\tau_1, \Gamma)) \vdash e_2 : \tau_2/\pi}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2/\pi} \qquad \text{DEFFUN}\ \frac{\Gamma; (x : \tau_1) \vdash e : \tau_2/\pi}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \xrightarrow{\pi} \tau_2/\pi_1}$$

$$\text{APPFUN}\ \frac{\Gamma \vdash f : \tau_1 \xrightarrow{\pi} \tau_2/\pi \qquad \Gamma \vdash e : \tau_1/\pi}{\Gamma \vdash f\ e : \tau_2/\pi} \qquad \text{OP AND CONST}\ \frac{}{\Gamma \vdash op : TC(op)/\pi} \qquad \text{PAIR}\ \frac{\Gamma \vdash e_1 : \tau_1/\pi \qquad \Gamma \vdash e_2 : \tau_2/\pi}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2/\pi}$$

$$\text{IFTHENELSE}\ \frac{\Gamma \vdash e : bool/\pi \qquad \Gamma \vdash e_1 : \tau/\pi \qquad \Gamma \vdash e_2 : \tau/\pi}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau/\pi} \qquad \text{VECTOR}\ \frac{\Gamma \vdash e : \tau/\ell}{\Gamma \vdash \langle\langle e \rangle\rangle : \tau\ \mathsf{par}\,/g} \qquad \text{APP VEC}\ \frac{\Gamma \vdash \langle\langle e \rangle\rangle : \tau\ \mathsf{par}\,/g}{\Gamma \vdash \$e\$: \tau/\ell}$$

Figure 2.13. Induction rules of the type system.

The rules of the type system are given in Figure 2.13. DEFFUN stores the locality induced by a function in the type arrow, but does not constrain the current locality (the rule is true for any π_1): the function itself can be defined anywhere. APPFUN then forces the current locality and the locality of an applied function to be the same, forbidding the use of functions that are not compatible with the current locality.

ENVTYPE specifies that when taking a type from the environment, it can be assigned any locality π. Together with LET-IN, this allows to use the results of a parallel computation at the desired locality (after using `proj`). This is needed for valid programs like `let` $x =$ `proj`$(\cdots)$`in` $\langle\langle$*expression using x locally*$\rangle\rangle$. $Gen(\tau, \Gamma)$ is defined in the usual way, except that it also generalises locality variables. The other rules are standard ML typing rules, and just propagate the current locality to sub-terms, which denotes the fact that the locality can only be changed by using the primitives, or applying functions that use them.

(b) Preventing Nested Parallelism

The type system defined above prevents crashes due to local execution of parallel code, but still allows the definition and projection of nested parallel vectors. Since we do not want to prevent the access to global values from local code (which is a very useful feature), and values that are members of parallel vectors can only be defined from the return values of local function, we choose to restrict the attribution of the ℓ locality to functions, depending on their return type. Note that this does not prevent values of type $\tau\,\mathsf{par}$ from being manipulated locally; however, since these can only be "opened" by the use of primitives that are forbidden in this scope, no harm can be done.

This amounts to adding a constraint on its locality and return type to the definition of a function. We write $\tau \triangleleft \Lambda$ to specify that the type τ must be acceptable in the locality Λ, *i.e.* that if $\Lambda = \ell$, τ must not contain parallel vectors. Locally acceptable types are a subset of all types, defined as

$$\dot{\tau} ::= \mathsf{Base} \mid \tau \xrightarrow{\pi} \tau \mid \dot{\tau} \times \dot{\tau}$$

The constraint $\tau \triangleleft \ell$ means that τ should belong to $\dot{\tau}$. We extend the effects π from the previous type system with sets of constraints as follows:

$$\pi ::= \Lambda[\mathrm{C}] \qquad \begin{array}{rcll} \mathrm{C} & ::= & \rho & \textit{row variable} \\ & \mid & \tau \triangleleft \Lambda; \mathrm{C} & \textit{constraint and rest of row} \end{array}$$

π now contains a set of constraints C along with the locality information. C is a row formed of constraints and terminated by a row variable, which is used for unification. With these extended effects, the function `fun`$x \to x$ would be typed $\alpha \xrightarrow{\delta[\alpha \triangleleft \delta; \rho]} \alpha$, which shows that the local use of the function is restricted by the type α. We can omit the terminating row variable when it is a free variable and write $\alpha \xrightarrow{\delta[\alpha \triangleleft \delta]} \alpha$. By allowing not to write the full row when it is empty (as in `fun`$x \to 0 : \alpha \xrightarrow{\delta} int$), this stays in accordance with former type definitions.

We extend the previous definition of type schemes by quantify row variables:

$$\sigma = \forall \alpha_1 \cdots \alpha_k \delta_1 \cdots \delta_l \rho_1 \cdots \rho_m . \tau$$

Substitution and free variable are also change accordingly and as a consequence of these new definitions, *Gen* also generalises row variables. We now need to define an equational theory on rows, to express the fact

that the same constraints can be expressed in different ways. First, order of constraints does not matter:

$$\gamma_1;\gamma_2;\mathrm{C} = \gamma_2;\gamma_1;\mathrm{C}$$

and several constraints can be ignored:

$$\begin{array}{rcl} \tau \lhd g;\mathrm{C} & = & \mathrm{C} \\ \tau_1 \xrightarrow{\pi} \tau_2 \lhd \Lambda;\mathrm{C} & = & \mathrm{C} \\ \mathsf{Base} \lhd \Lambda;\mathrm{C} & = & \mathrm{C} \\ \tau \lhd \ell;\tau \lhd \Lambda;\mathrm{C} & = & \tau \lhd \ell;\mathrm{C} \end{array}$$

Finally, the constraints can be distributed on subtypes until they either apply to type variables, or are of the form $\tau\ \mathsf{par} \lhd \Lambda$:

$$\tau_1 \times \tau_2 \lhd \Lambda;\mathrm{C} \quad = \quad \tau_1 \lhd \Lambda;\tau_2 \lhd \Lambda;\mathrm{C}$$

According to this theory, constraint rows can be reduced to rows containing only constraints of the form $\alpha \lhd \delta$, $\alpha \lhd \ell$, $\tau\ \mathsf{par} \lhd \delta$ or $\tau\ \mathsf{par} \lhd \ell$. The last case, $\tau\ \mathsf{par} \lhd \ell$, is unsatisfiable and should obviously be rejected ; the rules of the unification algorithm can, therefore, be limited to the three first cases.

The effects are built by unification during inference. The only rule that needs to be changed is DEFFUN, since we need to make the constraint on the return values of local functions explicit in the constraint row.

$$\frac{\Gamma;(x:\tau_1) \vdash e : \tau_2/\Lambda[\mathrm{C}] \qquad \mathrm{C} = \tau_2 \lhd \Lambda;\mathrm{C}'}{\Gamma \vdash \mathtt{fun}\ x \to e : \tau_1 \xrightarrow{\Lambda[\mathrm{C}]} \tau_2/\pi}\ \text{DEFFUN}$$

The safety property of typing can be stated as "well-typed programs do not go wrong". This property is obtained if the two following lemmas [108]:

Lemma 1 (*Subject reduction*)

The reduction of a well-typed BSML *expression preserves the typing that is: if* $\Gamma \vdash e_1 : \tau/\Lambda[\mathrm{C}]$ *and if* $e_1 \rightarrowtail e_2$ *then* $\Gamma \vdash e_2 : \tau/\Lambda[\mathrm{C}]$.

Lemma 2 (*Progress*)

Any well-typed BSML *program that is not a value can be reduced.*

The operational semantics $\rightarrowtail$ of a core-BSML (expressions and values) as well as the proofs of the above lemmas can be found in [73].

2.3.3 Imperative and Extended Features

(a) References

References can easily cause a loss of replicated consistency as shows the following code:

```
let a = ref false in
let x = ≪ a:=$pid$>0 ≫ in !a
```

a is a reference created globally, but which is written on locally. If it is then read back globally, we lose replicated consistency: This expression would have value **false** on processor 0 and **true** on the others. The problem occurs when references are written outside of the locality they were created in. For this reason, the type system annotates references with the locality they were created at:

$$\tau ::= \cdots \mid \tau\ \mathsf{ref}_\Lambda$$

The types of reference operators are then defined as:

$$\begin{array}{rcl} TC(\mathsf{ref}) & = & \forall\alpha\delta.\alpha \xrightarrow{\delta[\alpha \lhd \delta]} \alpha\ \mathsf{ref}_\delta \\ TC(\mathsf{store}) & = & \forall\alpha\delta.\alpha\ \mathsf{ref}_\delta \times \alpha \xrightarrow{\delta} \mathit{unit} \\ TC(\mathsf{value}) & = & \forall\alpha\delta_1\delta_2.\alpha\ \mathsf{ref}_{\delta_1} \xrightarrow{\delta_2[\alpha \lhd \delta_2]} \alpha \end{array}$$

But the **proj** can also introduce an error as shows the following code:

```
let f=proj ≪ let r= ref false in (fun x → r:=random_bool) ≫ in ((f 0) 0)
```

First a vector containing functions (closures) is create. Then these functions are extracted of the vector to generate the function f. That implies the creation of a new reference r since the values are serialised. But apply twice f to 0 implies the execution of random which should be only executed locally. A solution is to have **proj** change the types of references from local to global (which is actually what happens in the semantics and in the implantation). We do this by adding an intermediary type construct glob.

$$\tau ::= \cdots \mid \mathsf{glob}(\tau) \qquad\qquad TC(\texttt{proj}) = \forall\alpha\delta.\alpha\,\mathsf{par} \xrightarrow{g} int \xrightarrow{\delta} \mathsf{glob}(\alpha)$$

The functions that were defined by induction previously (φ and $\mathcal{L}$) propagate naturally to the arguments of ref_Λ and glob. The type $\mathsf{glob}(\tau)$ actually means a transformation on τ: it can be reduced using the following rules, until it only applies to type variables.

$$\begin{aligned}
\mathsf{glob}(\mathsf{Base}) &\Rightarrow \mathsf{Base}\\
\mathsf{glob}(\tau_1 \xrightarrow{\pi} \tau_2) &\Rightarrow \tau_1 \xrightarrow{\pi} \mathsf{glob}(\tau_2)\\
\mathsf{glob}(\tau_1 \times \tau_2) &\Rightarrow \mathsf{glob}(\tau_1) \times \mathsf{glob}(\tau_2)\\
\mathsf{glob}(\tau\,\mathsf{ref}_\Lambda) &\Rightarrow \mathsf{glob}(\tau)\,\mathsf{ref}_g
\end{aligned}$$

Note that $\mathsf{glob}(\tau\,\mathsf{par})$ and $\mathsf{glob}(\tau\,\mathsf{ref}_g)$ are invalid, because the argument of glob is a local return value, that must therefore follow the constraint $\triangleleft\ell$.

The reader may have noticed that glob propagates to the right-hand side of arrows although constraints do not, and it is not actually part of the data stored by values of these types. Since a local function can return any local reference present in its context, and which will in fact be communicated and globalised along with it, this is needed. It is at the cost of a small restriction in some cases: (**fun** i → a:=i>0) for example, has its return type restricted to ref_g if projected, forbidding its local execution.

(b) Exceptions

A concern with exceptions is that they may allow values to escape their scope. This has to be kept in mind when extending the type exn — which is, in OCAML, a variant type that can hold any other type. In particular, the global propagation of local exceptions has a power equivalent to that of **proj**; this means that values that can be raised locally must be restricted in the same way that local return values are. A simple and yet not overly restrictive solution is to keep exceptions free of parallel vectors and references, by adding a constraint to the types included in exn.

For our purpose here, we suppose that a base type exn is defined as well as a type exnset that holds associations between processor numbers and values of type exn. Values of type exn can be raised using the operator `raise`, which stops the current execution and gets up the stack until it meets an enclosing `try`…`with`$x \to e$ construct. This construct catches the exception, and triggers the behaviour specified in e. If no exception is raised inside, it is simply ignored.

The new constructs are typed with:

$$\text{TRYWITH}\ \frac{\Gamma \vdash e_1 : \tau/\pi \qquad \Gamma; x : \mathsf{exn} \vdash e_2 : \tau/\pi}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ x \to e_2 : \tau/\pi} \qquad\qquad \text{TRYPAR}\ \frac{\Gamma \vdash e_1 : \tau/g \qquad \Gamma; x : \mathsf{exnset} \vdash e_2 : \tau/g}{\Gamma \vdash \texttt{trypar}\ e_1\ \texttt{withpar}\ x \to e_2 : \tau/g}$$

$$TC(\texttt{raise}) = \forall\alpha\delta\rho.\,\mathsf{exn} \xrightarrow{\delta[\rho]} \alpha$$

The expression that triggers the exception and the one executed when an exception is caught must have the same type. TRYWITH has no special effect on locality, while TRYPAR can only be used globally.

(c) Other Features

The system described here does not, for the sake of simplicity, include record or variants types. Simple records pose no real problem: they are syntactic sugar over tuples.

Variants are more complicated, but the constructs and functions we defined can be extended to them without particular problems: a variant type follows $\triangleleft\ell$ if all of its alternatives do, and glob propagates likewise on all of them.

Although they have not been described here, modules are an important in the construction of OCAML programs; the type system has been designed to be compatible with them. Modules can be compiled

separately, and the inference system must be able to work on a partial program given summarised interfaces of the other modules used. All values from other modules are considered to be stored in the environment Γ. Effects on functions and locality annotations on references thus appear even on foreign modules. The point can cause some difficulty is the validation of the $\triangleleft\ell$ constraint on non-disclosed types from foreign modules. A possible solution is to compute the result of this constraint in advance within the foreign module and write it as an annotation in the interface.

But our type system lacks for some recent feature of OCAML: polymorphic records and variants, Generalised algebraic datatypes (GADT), explicit polymorphic type annotations, first-class value modules (Local modules and recursive modules) and objects has not been yet studied.

2.4 Implementation of Algorithmic Skeletons in BSML

From Publications 8

This section is taken from the following publications:

- *in [R1] and [W5] we describe the implementation of "flow skeletons" in* BSML *using the primitive of superposition;*
- *in [N1] we give an implementation of some "data-parallel" skeletons in* BSML.

2.4.1 What are Skeletons?

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, map, reduce, *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 [77, 109–111]. 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 [77].

In this light, having skeletons in BSML would have the advantage of the BSP pattern of communications (collective ones) and the expressiveness of the skeleton approach in the spirit of [112]. For our purposes and to have interesting benchmarks, we take for model the implementation of two kinds of skeletons [113]:

1. **Flow skeletons**; They manipulate a "flow" of data and mainly applied functions over these data; All these data flows (asynchronously) from machines to machines until a global condition occurs; We take for example, the OCAMLP3L [114] [[[11]]] skeletons language (P3L [115]'s set of skeletons for OCAML) and base them on our parallel superposition primitive;

2. **Data-parallel skeletons**; They mainly manipulate distributed data-structures as arrays, lists, trees, *etc.* and applying functions over the data of the structures.

There are already some advantages to using BSML-based skeletons: BSML can be used on a wide variety of communication libraries, such as BSPLIB, MPI and TCP/IP whereas for example, OCAMLP3L is currently stuck with TCP/IP. For sake of brievety, we do not present how all skeletons and utility functions are implemented; we only present the most interesting ones.

2.4.2 Implementation of Data-flow Skeletons: the OCamlP3L Language

(a) The Skeletons

Figure 2.14 [116] summarises the type of the skeletons. They work as follow.

The **seq** skeleton encapsulates an OCAML function f into a stream process which applies f to all the inputs received on the input stream and sends off the results on the output stream. **loop** computes a function f over all the elements of its input stream until a boolean condition g is verified.

The **farm** computes in parallel a function f over different data items appearing in its input stream. Parallelism is gained by having n independent processes.

```
val seq : (unit → 'a → 'b) → unit → 'a stream → 'b stream
val loop:('a→ bool)*(unit→ 'a stream→ 'a stream)→ unit→ 'a stream→ 'a stream
val farm : (unit → 'b stream → 'c stream) * int → unit → 'b stream → 'c stream
val pipe (|||) : (unit → 'a stream → 'b stream) → (unit → 'b stream → 'c stream) → unit → 'a stream → 'c stream
val mapvector : (unit → 'b stream → 'c stream) * int → unit → 'b array stream → 'c array stream
val reducevector : (unit → ('b * 'b) stream → 'b stream) * int → unit → 'b array stream → 'b stream
```

Figure 2.14. The types of the OCAMLP3L's flow skeletons.

```
val repl: 'a → int → 'a parList
val map: ('a → 'b) → 'a parList → 'b parList
val mapidx: ('a → int → 'b) → 'a parList → 'b parList
val zip: ('a → 'b → 'c) → 'a parList → 'b parList → 'c parList
val reduce: ('a → 'a → 'a) → 'a → 'a parList → 'a
val scan: ('a → 'a → 'a) → 'a → 'a parList → 'a parList
val dh: ('a → 'a → 'a) → ('a → 'a → 'a) → 'a parList → 'a parList
```

Figure 2.15. The types of the data-parallel skeletons.

The **pipeline** skeleton performs in parallel the computations relative to different stages of a function composition over different data items of the input stream.

mapvector computes in parallel a function over all the data items of a vector, generating the new vector of the results. The **reducevector** works in the same manner but doing an array folding with a binary operator as argument.

(b) Implementation of these Flow Skeletons

The combination of P3L's skeletons generates a process network. This network takes in input a stream of data. Then, each datum is transformed by the network independently of other data and finally the output is another stream of the same arity. In this way, these computations can be composed: supposing a n data stream, the execution of the network will be composed n times using the superposition: for each execution of a network, we use a counter that stores the placement of tasks and data in a round robin fashion; the P3L stream is implemented as a parallel vector of option values where only one processor keeps a non empty value — the data of the stream. The full stream is thus a list of these vectors.

Now, to produce the process network we recursively generate a BSML code from a skeleton expression in meta fashion — see [R1] for more details. For example, for the **pipeline**(s_1,s_2) skeleton, we directly compose them: the input of the resulting network is the input of the network of s_1 whose output is the input of s_2. If they are on distinct GPUs, we perform a sending function to connect the output of s_1 to the input of s_2. Note that for a BSP machine with **p** processors and a pipe of two sequential processes, the tasks would be distributed on all processors — we suppose a typical stream of more than **p** elements. Then, a single barrier would occur sending data from a processor to another one. This is clearly not the most efficient manner to execute the whole program but this is also clearly not an inefficient one.

The **mapvector**(n,s) skeleton is probably the most interesting one. The method is as follow. First, a new task is dynamically created for each element of the input vector of the stream and stored in a list of tasks, called ntasks. The BSML code for these tasks (produced by the sub-skeleton s) is generated from an inductive call. Then, once all the tasks created, their execution are *superposed* using the superposition. For each execution, the input processor of the network sends a data of the vector to the processor that has been dynamically designated to execute the sub-network. The parallelism arises from data being distributed over all superposed tasks. Finally, we gather the results to the network output processor.

(c) Benchmark

Our example is a parallel PDE solver which works on a set of subdomains, taken from [116]. On each subdomain it applies a fast Poisson solver written in C. The skeleton expression of the code is:

```
let PDE_solver = parfun (fun () →
 (loop ((fun (v,continue) → continue),seq(fun _ → fun (v,_) → v) ||| mapvector(seq(fun _→ compute_sub_domain),3)
    ||| seq(fun _→ projection) ||| seq(fun _→ bicgstab) ||| seq(fun _→ plot))))
```

and the coupling technique (and full equations) can be found in [116].

All the tests were run on the cluster LACL2. We present the benchmarks when the interface meshes match using randomly generated sub-domains — real life inputs are described in [116]. The principle of this extensibility test is as follow: increase number of processors as well as size of data. In this context,

for each input, one processor is associated with one sub-domain and the global domain is divided into 1, then into 2, 4, *etc.* sub-domains. Various manners of decomposing the global domain in a structured way are explored. The number of sub-domains along the axis is denoted by N_x (*resp.* N_y, N_z) and each sub-domain possesses approximately 50000 cells — time to sequentially decompose a sub-domain is approximately linear. The number of generated super-threads would be too big for our past implementation. Performances (minutes and seconds) of OCAMLP3L and BSML (using its MPI implementation) are summarised in the following table:

(N_x, N_y, N_z)	Nb procs	OCAMLP3L	BSML
$1 \times 1 \times 1$	1	20.56	21.29
$1 \times 1 \times 2$	2	24.06	27.63
$1 \times 1 \times 4$	4	24.78	28.23
$1 \times 1 \times 8$	8	25.05	28.97
$1 \times 1 \times 16$	16	26.53	30.67
$1 \times 2 \times 2$	4	20.78	25.14
$1 \times 2 \times 4$	8	24.45	28.36
$1 \times 2 \times 8$	16	25.56	29.84
$1 \times 4 \times 4$	16	26.89	29.89
$2 \times 2 \times 2$	8	25.88	27.21
$2 \times 2 \times 4$	16	27.89	32.75

As might be expected, OCAMLP3L is faster than our naive implementation but not much. Barriers slow down the whole program but bulk-sending accelerates the communications: in the P3L run there exists a bottleneck due to the fact that sub-domains are centralised and therefore the amount of communication treated by one process may cause an important overhead. In BSML, the data are each time completely distributed, which reduces this overhead but causes a loss of time. Our aim was not to beat OCAMLP3L, whose implementation is far more complicated than ours but have both BSML and OCAMLP3L and to not be stuck with TCP/IP as OCAMLP3L (currently) is.

2.4.3 Implementation of Some Data-parallel Skeletons

(a) A Set of Data-parallel Skeletons

Figure 2.15 summarises the set of data-parallel skeletons proposes in [92]. They are working on distributed "lists" and their functional semantics is obvious. Except for dh which is a a *Distributable Homomorphism* — while reduce and scan are like the MPI's collective operations. This skeleton is used to express a special class of divide-and-conquer algorithms. $dh \oplus \otimes l$ transforms a list $l = [x_1, \cdots, x_n]$ of size $n = 2^m$ into a result list $r = [y_1, \cdots, y_n]$ of the same size, whose elements are recursively computed as follows:

$$y_i = \begin{cases} u_i \oplus v_i & \text{if } i \leq \frac{n}{2} \\ u_{i-\frac{n}{2}} \otimes v_{i-\frac{n}{2}} & \text{otherwise} \end{cases}$$

where $u = \mathbf{dh}\ \oplus \times [x_1, \ldots, x_{\frac{n}{2}}]$, *i.e.* **dh** applied to the left half of the input list l and $v = \mathbf{dh}\ \oplus \times [x_{\frac{n}{2}+1}, \ldots, x_n]$, *i.e.* **dh** applied to the right half of l. The **dh** skeleton provides the well-known *butterfly* patten of computation which could be use for implementing many problems.

(b) Implementation of these Data-parallel Skeletons in BSML

These skeletons manipulate lists which are in fact distributed on the processors. Notice that they are persistent (no skeleton modifies in place a list). That allows us to implemented lists as vectors of arrays with their (constant) global sizes. For example:

```
type 'a flow = 'a array par * int

let repl a n = let p=bsp_p in
 (≪  if pid<n mod p
      then Array.create (n/p+1) a
     else Array.create (n/p) a≫ ,n)

let map f (fl,n) = (≪  (Array.map f) $fl$≫ ,n)
```

The skeleton **repl** create **p** arrays with the same length (modulo 1) following by the size n. With this distribution of the elements, the skeletons are easy to implement. Most of them are a simple asynchronous application of a sequential function on the sub-lists.

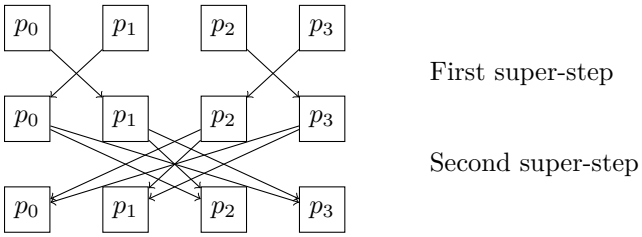
The **dh** skeleton is the harder one. [92] propose to implemented in a shared memory parallelism by first gather the elements in a server and them used parallel threads. Without lost of generality and for simplicity of the presentation, we supposed that $\mathbf{p} = 2^q$ — else, processes are distributed in a round-robin

```
(* super_mix: 'a par*'a par → 'a par *)
let super_mix m (v1,v2) = ≪ if $pid$<=m then $v1$ else $v2$≫

(* dh : ('a → 'a → 'a) → ('a → 'a → 'a) → 'a flow → 'a flow *)
let dh oplus omult (fl,len) =
 let rec tmp n1 n2 n vec =
  if n=1 then vec else
   let n'=n/2 in
   let n1'=n1+n' and n2'=n1+n'−1 in
   let vec'= super_mix (n1'−1) (super (fun () → tmp n1 n2' n' vec)
                                      (fun () → tmp n1' n2 n' vec)) in
   let send= put ≪ if pid<n1' then (fun dst → if dst=($pid$+n') then Some $vec'$ else None) else (fun dst →
if dst=($pid$−n') then Some $vec'$ else None))
 in ≪ if $pid$<n1' then match ($send$ ($pid$+n')) with
                          Some b → Array.map2 oplus $vec'$ b
                        | None → $vec'$
                    else match ($send$ ($pid$−n')) with
                          Some b → Array.map2 omult b $vec'$
                        | None → $vec'$≫
 in (tmp 0 (bsp_p−1) bsp_p ≪ local_dh oplus omult (Array.copy fl)≫ ,len)
```

Figure 2.16. Implementation of the **dh** skeleton in BSML.

fashion on the true processors. Figure 2.16 gives the BSML code of the distributed implementation of **dh** which looks like the scan algorithm — using the superposition.

First, each processor locally computes a partial butterfly with its own data (local_dh). Then we used a recursive function that compute the full butterfly by merging the elements in a logarithmic number of super-steps. If we image processors as an array, at each super-step s, processors are cut into $\mathbf{p} - 2^s$ parts where each sends its elements to others of the neighbour part as scheme here for 4 processors:



Suppose, the cost of operator $\oplus$ (resp. $\otimes$) has a constant execution time noted $c_\oplus$ (resp. $c_\otimes$) and a initial list which contains $2^m = \mathbf{p} \times 2^n$ elements (*i.e.* $m = n + q$) each having a constant size s, we can deduce the following BSP cost: $\log(\mathbf{p}) \times (2^n \times s \times \mathbf{g} + \mathbf{L} + n \times 2^n(c_\oplus + c_\otimes))$

(c) Benchmarks

As first example, we consider the solution of the "Tridiagonal System Solver" (TDS) of equations [92]: $A.x = b$ where A is a $n \times n$ sparse matrix representing coefficients, x a vector of unknowns and b a right-hand-side vector. The only values of matrix A unequal to 0 are on the main diagonal, as well as directly above and below it. A solution of this problem using the above data-parallel skeletons is presented in [92]. It mainly represents the TDS as a list of rows and the using this list representation, the algorithm can be expressed as follows (see [92] for more details): $tds\, m = \mathbf{map}\, \pi_1\, (\mathbf{dh}\ \oplus\ \otimes\, (\mathbf{map}\, triple\, m))$ with appropriate $\oplus$ and $\otimes$ operators. The TDS problem is mainly implemented using the **dh** skeleton. The sequential complexity of operator $\oplus$ (resp. $\otimes$) has a costly constant execution time noted $c_\oplus$ (resp. $c_\otimes$). If we suppose a initial matrix A which contains $\mathbf{p} \times 2^n$ data, each data have a constant size s, we can deduce the following BSP cost: $\log(\mathbf{p}) \times (2^n \times s \times \mathbf{g} + \mathbf{L} + n \times 2^n(c_\oplus + c_\otimes))$

As second example, we consider a "Fast Fourier Transform" (FFT) that is a transformation of a list $x = [x_0, \ldots, x_{n-1}]$ of length $n = 2^m$ yields a list whose ith element is defined as:

$$(FFT\, x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$$

where ω_n denotes the nth complex root unity $e^{2\pi\sqrt{-1}/n}$. As above, a solution of this problem is presented in [92]. FFT can be expressed as follow:

$$\begin{array}{lcl}(FFT\, l) & = & \mathbf{let}\, \Omega = \mathbf{scan}\ +\ 1\, (\mathbf{repl}\, (\omega\, n)\, \frac{n}{2}) \\ & & \quad \mathbf{in}\ \mathbf{map}\, \pi_1\, (\mathbf{dh}\ \oplus\ \otimes\, (\mathbf{mapidx}\, triple\, l))\end{array}$$

with appropriate $\oplus$ and $\otimes$ operators — see [92] for more details. The sequential complexity of operator $\oplus$ (resp. $\otimes$) has a constant execution time noted $c_\oplus$ (*resp.* $c_\otimes$) and we note c_ω^n the linear complexity of computing ω^n. If we suppose a initial list which contains $\mathbf{p} \times 2^n$ complexes, (each have a constant size

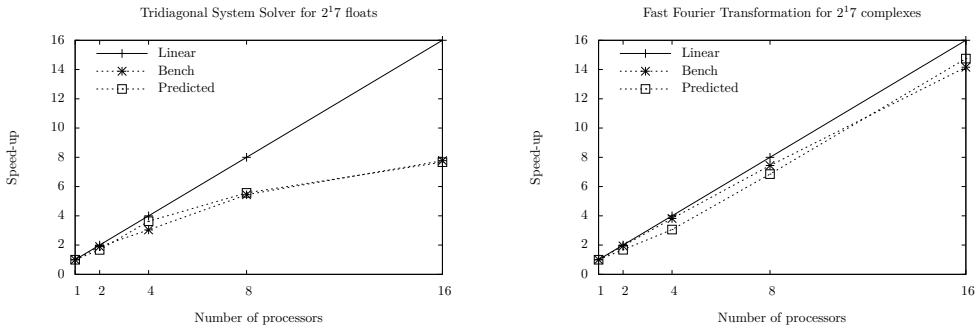


Figure 2.17. Benchmarks of the TDS and FFT problems using data-parallel skeletons.

s), we can deduce the following BSP cost (cost of the two skeletons):

$$\log(\mathbf{p}) \times (s \times \mathbf{g} + \mathbf{L} + c_{\omega}^{n} + 2^{n} \times s \times \mathbf{g} + \mathbf{L} + n \times 2^{n}(c_{\oplus} + c_{\otimes}))$$

Figure 2.17 gives the predicted and measured performances (using the PUB implementation on LACL1) of the TDS problem (in the left) and the FFT problem (in the right) using our implementation of the data-parallel skeletons. We gives the speed-up for a particular n. Estimated times were done by insert the measured BSP parameters $\mathbf{g}$ and $\mathbf{L}$ into the above BSP costs formulae.

For the two problems, measured and predicted times are close. This is due to the fact that our program are numerical ones and that OCAML is know to have good predicted execution time of its programs if the garbage collector is not used too much: we only used array of floats/complexes and thus no complex data-structures. But that is hided to the programmer which only see composition of high-level skeletons.

Speed-up is very good for the FFT problem. This is not a surprise since it is a well known good problem for parallelism. For the TDS problem, the over-cost of communications/synchronisations make the speed-up classically decrease when we increase the number of processors. But the two problems scalable well as the BSP cost described it. It is very encouraging because ML's programs have the absurd reputation of being slow and inefficient. Note that we did not compare the performance of these programs with hand-written ones or C ones. The performance will obviously be much better but at a price of more potential errors in programs.

2.5 Mechanised Correctness of BSML Programs

From Publications 9

This section summarises the work of [R8].

The type system we have defined prohibits unsafe programs to be executed. But it is not sufficient for programs which need *formal proofs* of the algorithms or for some libraries which need to be certified, *i.e.* programs must be shown to compute what is expected from them, that is prove their *correctness*.

In the proof assistant COQ[2], there is a *constructive interpretation* of proofs, *i.e.* proving a formula implies explicitly constructing a typed λ-term — this is the famous Curry-Howard isomorphism. In this way, the specification of a program can be represented by a logical formula and the program itself can be extracted from the constructive proof of the specification. For example, this is the approach taken in [117] (CompCert project) to mechanised generate a correct C compiler.

The functional approach of BSML allows the re-use of suitable techniques for formal proof from functional languages because a small number of parallel operators is needed for an explicit parallel extension of a functional language and it keeps the advantages of the BSP model. This is the approach we have choose for mechanised correctness of BSML programs in [R8]: we have built a certified library of common functional BSP programs. To represent our parallel language and to have a specification-extraction of functional BSP programs, we have chosen a classical approach: an *axiomatization* of the parallel primitives as show in Figure 2.18 for two of the operators — **put** and the **mkpar** of the past primitive; we use the past primitives because introduces the new syntax in COQ were less stable. Thus, operations are given by *parameters* and do not depend on the implementation — sequential or parallel.

[2]Some other systems such as HOL or PVS offer similar possibilities.

```
Parameter bsp_p: unit->Z.
          Vector: Set->Set.
          mkpar: forall T:Set, (Z->T)->(Vector T).
          put: forall T:Set, (Vector (Z->(option T)))

Parameter att: forall T:Set, (Vector T)->forall (i:Z), 0<= i<(bsp_p tt)->T.
Axiom good_bsp_p: 0<(bsp_p tt).

Axiom mkpar_def: forall (T:Set) (f:Z->T) (i:Z) (H:0<= i<(bsp_p tt)), (att T (mkpar T f) i H)=(f i).

Axiom put_def: forall (T:Set) (Vf:(Vector (Z->(option T)))) (i:Z) (H:0<= i<(bsp_p tt)),
 ((att (Z->(option T)) (put T Vf) i H)=(fun j:Z => match (within_bound j) with
      left H1 => ((att (Z->(option T)) Vf j H1) i)
    | right _ => None end)).

Axiom at_H: forall (T:Set) (v:(Vector T)) (i:Z) (H1 H2:0<= i<(bsp_p tt)), (att T v i H1)=(att T v i H2).
```

Figure 2.18. Axiomatization of some of the BSML primitives.

The process number, `bsp_p()`, is naturally an integer given by the parameters. An axiom indicates that it is greater than 0. Parallel vectors are indexed over type `Z` (COQ's integer), starting from 0 to the constant process number. They are represented in the logical world by an abstract dependent type `Vector T` where `T` is the type of its elements. This abstract type is manipulated only by one of the parameters. `at` is an abstract "access" function for the parallel vectors. It gives the local value contained in a process and it would be used in the specification of programs to give the values contained in the parallel vector result. It has a dependent type to verify that the number `i` is a valid process number.

Then, the specification of the BSML primitives follow their semantics. For example, the `put` operator, is axiomatising with `put_def`. It transforms a functional vector to another functional vector which guides communication using the functional parameter `j` to read values from distant processes. The parameter `j` is tested with the function `within_bound` which tells whether an integer is a valid process number or not and which gives the proof. If it is valid, the value on the process `i` is read on process `j` with the access function (`j` is a valid process number; the proof is given by `within_bound`). Otherwise, an empty constant is returned.

In [R8], we have used this axiomatization to develop certified functional BSP programs that mainly correspond to the functions which are provided as a standard library when installing BSML. We have mechanically verified broadcasting/total exchange/scatter/gather functions, a one-phase reduce function and the PSRS sorting — which has been the longer one. Finally, to extract COQ specifications into BSML programs, we have just to tell the COQ system that the parameters are the BSML primitives.

Our approach suffers from several drawbacks. First, BSML primitives use OCAML's integers and not COQ's `Z`. That forces to convert the extracted programs that can introduce a mistake. Second, we have not prove the consistency of these axioms within COQ. Third, proving programs in COQ is hard for non-specialists. To address the two first problems, the authors of [86] and [118] have used the "type class" feature of COQ to abstract the BSML primitives: they are now parameters of the programs and depend themselves on an implementation of integers. In this way, an extracted program depends on an implementation of the primitives and of the integers. A certified implementation in COQ of the BSML primitives using vectors has been done to prove the consistency of the primitives. The only lack of safety is that we do not have any proof that the parallel implementation of the BSML primitives is correct. Finally, in [118], the authors have tested their method and benchmarks an extracted program for solving the "Heat Diffusion Simulation" problem. In addition, in [73] (followed by [88]), the authors have implemented in COQ a generic tree skeleton (close to the one of [119] and part of the SKETO library [[12]] [120]) using COQ and an axiomatization of the BSML primitives. In [121], the same axiomatization has been used for the correctness (and to extract certified BSML programs) of data-parallel list skeletons — close to the ones presented in this chapter.

The idea of using the extraction feature of COQ for parallel programs with purely functional semantics (as BSML and skeletons) has also been used in [122] for the verification of the "Heat Diffusion" problem written with a skeleton library. As BSML primitives, this set of skeletons has been axiomatized as a type class in COQ. In addition, a vector implementation in COQ has also been done for consistency. Independently, in [123], the authors perform the same work, but for the MAPREDUCE framework. In [124], authors propose to use the Bird-Meertens formalism in COQ to systematically have MAPREDUCE programs.

2.6 Related Work

2.6.1 Programming Languages and Libraries for BSP Computing

MPI programs that only use *collective operations* are close to BSP ones [25]. [9] argues that most MPI programs can be thought as a sequence of collective patterns: most asynchronous communications are used to simulate a collective pattern that is not present in the standard MPI. [125] proposes an automatic tool (not yet fully implemented) to transform asynchronous communications into some calls of collective operations.

There are "many" libraries for BSP computing. Historically, the first one was the BSPLIB [78] for the C programming language. Based on this library, for graph manipulation, CGMLIB [56] [[13]] has been developed. It also limits the BSP model by simplifying schemes of communications: only total exchanges are mainly authorised. The BSPLIB has also been re-implemented using MPI in BSPONMPI [[14]]. The BSPLIB has also been extended in the Paderborn University BSP library (PUB) [91] by adding subset synchronisations, high-performance operations and migration of threads (only using TCP/IP). The PUB library has also been adapted to the JAVA language and grid computing in [126]. In [127], the authors have extended the BSPLIB to execute programs in heterogeneous systems.

For the JAVA language, different libraries exist. To our knowledge, the first one is [128]. There is also BSP-Core [29] (which also works for C ++) [[15]] which aims to provide a BSP library for multi-core technologies. There is also [129] which also provides *code migration* — using the facilities of JAVA to serialise objects. More recently, a library with scheduling and migration on grid environments of BSP threads has been designed in [72] — the scheduling is implicit but the migration can be explicit. But the most known library is HAMMA [130] [[16]]. It ables BSP's communications and the implementation is provided by a "MAPREDUCE like" framework of the foundation APACHE. A comparison with the Google's MAPREDUCE language [131] is described in [132]. The author notes that some graph algorithms could not be efficiently implemented using MAPREDUCE. Notice that another Google's language call PREGEL [[17]] (without any public implementation) is also used by this company to perform BSP computations on graphs. We can also highlight the work of NESTSTEP [133] [[18]] which is C/JAVA library for BSP computing, which authorizes nested computations in case of a cluster of multi-core but without any safety.

For the GPU architectures, a first BSP library was provided in [30] [[19]]. It is mainly the primitives to remote access memory of the BSPLIB/PUB. The BSML primitives were adapted for C ++ in BSP ++ [134] [[20]]: this library provides nested computation in the case of a cluster of multi-core (MPI +OPEN-MP). The BSML language also inspired BSP PYTHON [135] [[21]], which will be used in Chapter 4. BSML also inspired BSP-HASKELL [136]. The library [28] provides collective operators in a BSP fashion for multi-core architectures.

2.6.2 Parallel Programming

There are really many parallel languages or parallel extensions of a sequential language (functional, iterative, object-oriented, *etc.*). It would be too long to list all of them. We chose to point out those that were the most important to our mind. Notice that, except in [137], there is a lack of comparisons between parallel languages. But it is hard to compare them since many parameters have to be taken into account: efficiency, scalability, expressiveness, *etc.*

(a) Parallel Functional Languages

Two nice introductions (with many references) to parallel functional programming can be found in [138] and [8]. They have been used as a basis for the following classification — with some updates.

The authors explain three reasons to use functional languages in the parallel programming. First, they ease the partition of a parallel program (task decomposition). Second, most of them are deadlock free: the value is independent of the evaluation order that is chosen and thus any program that delivers a value when run sequentially will deliver the same value when run in parallel. Third, they have a straightforward semantics which is great for debugging. Testing and debugging can be done on a sequential machine: functional programs have the same semantic value when evaluated in parallel as when evaluated sequentially.

Data-parallel Languages. To our knowledge, the first important *data-parallel* functional language was NESL [139] [[22]]. This language allows to create specific arrays and nested computations within these arrays. The abstract machine (or the compiler) is responsible for the distribution of the data and computations over the available processors. Two extensions of NESL for ML programming are NEPAL [140] [[23]]

and MANTICORE [141] [[24]]; the latter is clearly a mix between NESL and Concurrent ML [142] [[25]] — Concurrent ML is a language of the creation of asynchronous threads and send/received messages in ML.

We can also cite the following data-parallel languages. First, SAC (Single Assignment C) [143] [[26]] is a lazy functional language (with a syntax close to C) for array processing. Some higher-order operations on multi-dimensional arrays are provided and the compiler is responsible for generating an efficient parallel code. An extension of the famous lazy functional language HASKELL is Data Parallel HASKELL [144] [[27]]. It allows to create data arrays that are distributed across the processors. And some specific operations permit to manipulate them.

The main drawback of these languages is that cost analysis (for comparing algorithms) is hard to do since the system is responsible for the data distribution.

Explicit process creation. In this category we have two extensions of HASKELL: EDEN [145] [[28]] and GPH (Glasgow Parallel HASKELL) [137, 146] [[29]]. Both extend HASKELL with a small set of syntactic constructs for explicit process creation. Their fine-grain parallelism, while providing enough control to implement parallel algorithms efficiently, frees the programmer from the tedious task of managing low-level details of communications. Those communications uses lazy shared data. Processes (threads) are automatically managed by sophisticated runtime systems for shared memory machines or distributed ones. Note that both languages are also extended with algorithmic skeletons [147].

As above, cost analysis of the programs is hard to do for these languages and, sometimes, runtime fails to distribute correctly the data [148]; this introduces too much communications and thus a lack of scalability. Another distributed language is HUME [149] [[30]]. The main advantage of this language is that it is provided with a cost analysis of the programs for real-time purpose. But this analysis limits expressiveness.

Algorithmic skeletons and their implementation. As described previously, skeletons are patterns of parallel computations [110, 150]. They can be seen as high-order functions that provide parallelism. They fall into the category of functional extensions [151] — following their semantics [152]. Most skeleton libraries extend a language (mostly JAVA, HASKELL, ML, C/C ++) to provide those high level primitives.

Currently, the most known library is the Google's MAPREDUCE [131]. It is a framework to process embarrassingly parallel problems across huge datasets — originally for the page-ranking algorithm. Different implementations for JAVA or C exist. But only two skeletons are provided which limit expressiveness. [153] provides a set of flow skeletons for C ++. Templates are used to provide an efficient compilation of the programs: for each program, a graph of communicating processes is generated and is then transformed into a classical MPI program. Templates has also been used in [154].

For JAVA, many libraries exist such as the ones of [155, 156] and [157]. The latter has been extended for multi-core architectures [158]. A study of how to type JAVA's skeletons has been done in [159]. The authors note that some libraries of data-flow skeletons use a unique generic type for the data (even if it is an integer or a string), which can cause a clash of the JVM. They explain how to avoid this problem, using a simple type system.

[77] describe how to add skeletons in MPI (the eskel library), as well as some experiments. It also gives convincing and pragmatic arguments to mix message passing libraries and skeleton programming. We think that using OCAML in parallel programming (HPC applications) is not a wrong choice, since the generated code is often very competitive with the C counterparts. Some benchmarks of an OCAML implementation of data-flow skeletons for a numerical problem is described in [116]. But, the implementation currently sucks to TCP/IP. The first study of how to integrate flow skeletons and data-parallel ones was in [113]. In our work, we provide both, since these skeletons are implemented using BSML— a part of the ideas were taken from [112] and HDC [160].

Implementing skeletons using a BSP library was first done in [161] and application of the BSP cost prediction was also done in [162]. In both works, a C library was used and the set of skeletons was smaller than our. Another BSP implementation of skeletons has also been done in [163]. Using ML, our implementation is safer. [164] describes an implementation of a data-flow skeletons set using OCAML +MPI. It has been used for an imaging application [165]. Two older skeleton libraries for ML have been designed in [166] and [167].

Our current set of data-parallel skeletons works on lists. Thinking of other data structures is natural. This is the case of the SKETO library [119, 168]. The authors of SKETO tried to provide a minimal set of skeletons in which the programmer can derive other skeletons [169]. Currently our set of skeletons manipulates lists of constant sizes. A first work with dynamic sizes of lists has been done in [170]. Note that an implementation for multi-cores architectures has been done in [171].

To finish, all these skeleton libraries have the same drawback: they suppose that the sequential parts of the skeletons (usually named functional or business code) cannot fail. In this case, the whole machine

can fail: there is no exception handling. A first work on this subject can be found in [172]. Using our BSML's implementation, we can benefit from the exception handling of BSML to catch this case.

Data-flow and Coordination languages. Coordination languages [173] (and data-flow languages, which are close) add primitives to a sequential language in order to describe how to coordinate processes over a parallel architecture; they mainly generate a graph of the data flowing. One of the ancestor is CALIBAN [174] which allows the declarative description of a processes network and the mapping of processes to processors. The compiler generates a code for each process and their interconnection, which is thus statically determined. This is an extension of a purely functional language.

One of the most popular is SISAL (Streams and Iteration in a Single Assignment Language) [175] [[31]]. It is close to most statement-driven languages, but variables should be assigned once. This allows the compiler to identify easily the inputs and outputs. Currently, the two main approaches are LINDA [176] and PCN (Program Composition Notation) [177], which have been implemented in many programming languages. Therefore, both languages are "notations" which are inserted in sequential languages (C/C ++, LISP, JAVA) to generate and manage the mapping of processes to processors. In both languages, the notation can be seen as composition of functions/processes. Last, a complex runtime system needs to adequately synchronise concurrent memory accesses of the processes.

Distributed functional languages. In front of parallel functional languages, there are many concurrent extensions of functional languages such as ERLANG [[32]], CONCURRENT CLEAN or JOCAML [178] [[33]]. The latter is a concurrent extension (based on the join-calculus) of OCAML, which added explicit generation of parallel processes and a way to distribute them across processors using "resources". Using JOCAML, processes synchronisation is achieved using specific patterns called join-patterns.

ALICE ML [179] [[34]] adds what is called a "future" for communicating values. A future is a placeholder for an undetermined result of a concurrent computation. When the computation delivers a result, the associated future is eliminated by globally replacing it by the result value. The language also contains "promises" that are explicit handles for futures. SCALA [[35]] is a functional extension of JAVA which provides concurrency, using the actor model: mostly, creation of agents who can migrate across resources. OZ-MOZART [180] [[36]] is a multi-paradigm language (originally functional) with explicit creation of threads. Communications are performed using "ports". It also provides data-flow computation over lazy lists. Unfortunately, the execution speed of a program produced by the MOZART compiler is very low.

For HPC, all these languages have the same drawbacks: they are not deadlock and race condition free; furthermore, they do not provide any cost model for reasoning of algorithmic optimisations.

(b) Parallel Libraries and Imperative Programming

There is many parallel libraries but the two most used are certainly MPI and OPEN-MP. Both are working under C/FORTRAN — even if we can find bindings for JAVA, OCAML and other languages. MPI contains many operations such as asynchronous sending values, collective operations, parallel I/O accesses, *etc.* for distributed architectures. OPEN-MP has been design for working on shared-memory architectures and allows creation/synchronisation of threads. It mainly works using annotations in the code — *e.g.* indicates which loop can be achieved in parallel.

The "parallel loop paradigm" (where the compiler automatically distributes the computations) has given rise to many extensions of C and JAVA. We can cite SPLIT-C [181] [[37]], TITANIUM and UNIFIED PARALLEL C [182] [[38]], CILK [[39]], FORTRESS [[40]], *etc.* Mainly, the management of the processes and the implementation are the only differences for these languages.

ZPL [183] [[41]] (and its ancestor ORCA [184] [[42]]) is an array programming language. The programmer gives instructions (with specific primitives) to manipulate the arrays (*e.g.* to copy a value in the cells above) and the compiler automatically distributes the computations and the data. It is close to SAC but it is not a data-parallel language. This kind of operations can also be found in CO-ARRAY FORTRAN [[43]].

CILK [185] is an extension of C where the programmer can spawn a procedure like another thread of execution. Communications are performed using shared variables and synchronisations. OCCAM-PI [[44]] was designed following the Communicating Sequential Processes (CSP) process algebra and the π-calculus. The business code can be either HASKELL or C code. They appear in π-terms using a specific construction.

CHARM++ [186] [[45]] is an extension of C ++ based on a migratable-objects programming model. The programmer decomposes the program into a large number of "chare", which can migrate and interact with each other via asynchronous method invocations. The system maintains a "work-pool" consisting of seeds for new chares, and messages for existing chares.

2.6.3 Extensions and Applications

(a) Explicit Vector Notation

As some readers must have pointed out, the notation $\ll$ e$\gg$, which builds a parallel vector, is close to constructions like "e1 par e2" in GPH [137, 146] (if it is used $\mathbf{p}-1$ times), or to creation of processes in EDEN [145]. However the meaning of these constructions differs. Fine-grained parallelism introduced by GPH's "par" takes two arguments that are to be evaluated in parallel. The expression "e1 par e2" has the same value as "e2". Its dynamic behaviour consist in indicating that "e1" could be evaluated by a new parallel thread, with the parent thread continuing evaluation of "e2". Threads are then distributed on the processors at run-time. Communications are implicit by the share of variables.

In our case, parallelism is explicit (as well as the communications and the data distribution) and especially it is prohibited to nested parallelism to optimise performances of the implementation [73]. BSML is clearly a lower level programming language compared to GPH or algorithmic skeletons. Nevertheless, it provides a realistic cost model and is well adapted to write coarse-grain algorithms. Moreover, it can be used to implement higher-order parallel functions that will then be used as algorithmic skeletons.

(b) Imperative and I/O Features

Our imperative features (references and I/O operations) have been designed to be fully deterministic. The authors of [187] argue that it is better to have undeterministic I/O operations, cause it is more realistic and the programmer can control this undeterministic using specific procedures. We believe that safety is more important and "controlling chaos" is a tedious work for many programmers.

PLASMA [[46]] is a recent MAPREDUCE extension of OCAML with a distributed key/value database and NFS (Network File System) files. It provides an automatic fault-tolerant management of the files. It could be a valuable basis for an implementation of BSML I/O operations.

With few exceptions, previous authors focused on uni-processor external memory models. The *Parallel Disk Model* (PDM) introduced by Vitter and Shriver [188] is used to model a two-level memory hierarchy. In BSML, the model is mainly the one of [95] which provides parallel disks for a parallel architecture.

Some other parallel functional languages like SAC, EDEN or GPH, offer some I/O features but without any cost model. I/O operators in SAC have been written for shared disks without formal semantics and the programmer is responsible for undeterministic results of such operations. In EDEN and GPH, the safety and the confluence of I/O operators are ensured by the use of *monads*. These parallel languages also authorise processors to exchange channels and give the possibility to read/write to/from them. It increases the expressiveness of the languages but decreases the cost prediction of the programs. Moreover, too many communications are hidden and it also makes the semantics difficult to write.

In [189], the author presents a dynamic semantics of a mini functional language with a call-by-value strategy but I/O operators do not work on files. The semantics used a unique input entry (standard input) and a unique output. The authors of [190] have developed a language to reason about concurrent pure functional I/O. They prove that under certain conditions the evaluation of this language is deterministic. But the files are only local files and no formal cost model is provided. In [191], the authors have implemented some I/O operations to test their models but in a low level language. In the same way, [192] describes an I/O library of an EM extension of its cost model which is a special case of the BSP model, but also for a low level language.

(c) Parallel Pattern-matching

Our idea to parallel patterns for parallel vectors comes from the patterns of arrays of rewriting languages such as TOM [193] [[47]] (for JAVA and C) and MGS [194] [[48]] — for OCAML. Both use regular expressions as patterns for arrays. For instance, 0*+1* matches an array containing 0 or 1 everywhere.

Our syntax for pattern matching of vectors is close to regular expressions but is more powerful without being harder to read, due to the fixed size nature of the vectors and to the fact that there is no real order between elements in the vectors: each value in the vector is just a value contained by one processor only.

(d) Parallel Exceptions

There is little literature about parallel exceptions. In [195, 196], all the authors use techniques coming from concurrent languages to manage exceptions. Mainly, the exception handling consists in stopping processes that do not catch the exception and propagating a specific exception when a process stops abnormally. This is the case in the work of [197], in Concurrent Haskell, where an exception can propagates

over different processes which also generates non-deterministic results — in BSML, our exception handling is fully deterministic. This is also the case for the exceptions in ARGUS [198], where a subset of processors can select the exception to be treat.

In MANTICORE, exceptions follow a sequential semantics, which means that for a given expression, exceptions are handled in the sequential evaluation's order. When multi-processes raise different exceptions, a complex order is given to prioritise the treatment of the exceptions. We believe that the exception management in BSML is simpler.

(e) Numerical Applications in OCaml

OCAML has not been designed for numerical applications but rather for symbolic computing. Nevertheless, some works about it exist. [199] is the first book that gave some useful examples of the use of OCAML for scientific computing. It shows the advantage of a polymorphic, safe, efficient language for this community. We should also note OCAMLFLOAT [[49]], which is an interface to the LAPACK and BLAS libraries. It aims to improve the clarity and efficiency of numerical algorithms: this library allows to build a matrix using sub-matrices and provides a common interface for matrices of floats and complex numbers. But there is no interface for other kind of data.

For OCAML, we found three libraries to manipulate matrices: PSILAB [[50]] and LACAML [[51]] which both use LAPACK to provide matrices operations. PSILAB is very convenient to print capacities for matrices. To our knowledge, none of them offers parallel operations, true generic interface and safety/portability of execution all together. The matrix library SCIPY [[52]] for the PYTHON language also misses these properties.

2.7 Lessons Learnt From this Work

Designing and implementing a parallel functional language allows you to learn many things: type systems, parallel libraries, parallel algorithms and cost models, theorem proving, CPS, low level details of the target language (OCAML), *etc.*

The first lesson is that, each time the primitives are changed, it is necessary to change the type system since new kinds of errors can be added. Our first type system has been designed before the extensions described above. That revealed to be a wrong choice. I believe it is better to start increasing the expressiveness (and the possibilities of the language) before seeking safety immediately, even if safety has always to be kept in mind.

Second, applications and tests are needed immediately. Parallel model-checking (see next chapter) is a good test, since complicated algorithms are used. The feedback of students is also highly instructive: they are not good enough programmers to understand all the subtleties in a programming language and therefore do all possible and imaginable errors. Moreover, it is a good way to see if they can understand their errors and if they are able to correct them.

Third, the design of our modular implementation has been fruitful, because this allows us to achieve a much simpler and safe implementation. This design was possible only after a semantic study. We believe that many parallel languages with complicated runtime systems would have simpler and safer implementations if semantics studies had been performed first.

3 Deductive Verification of BSP Algorithms

This chapter summarises and extended the work of [C9], [C8], [C10] and [W4]. This work was done within the framework of the doctoral thesis of Jean Fortin — defence in March 2013.

In this chapter, we will present a tool for deductive verification of BSP algorithms. This tool call BSP-WHY is an extension (for BSP computing) of the back-end condition generator WHY. It thus takes an annotated algorithm and generates proof obligations for proving the correctness of the algorithm. It mainly works by transforming parallel algorithms into equivalent sequential ones.

Some of the most important details of the transformation are given — the formal transformation will be fully available in the doctoral thesis of Jean Fortin. Some simple examples illustrate this chapter. We also present the most important rules of the mechanised semantics of a core-calculus of this language which has been done in COQ.

Contents

3.1 Generalities and Background

3.1.1 The Correctness of Parallel Programs

(a) Importance and Needs (Reminder)

It is well known that programs too frequently stop working, and must be reprogrammed to clear bugs and logical errors. Developing tools and methods to finds these bugs or to help programmers having "from the start" (or at a given time of the development) correct programs is an old but still pertinent area of research. For sequential (iterative, functional, object or real-time) algorithms and programs, many methods exist: programming throw a theorem proving, B, model-checking, test, deductive verification, abstract interpretation, *etc.* But there is clearly a lack for parallel programs. Which is a shame because

these programs are extensively used in scientific calculations[1] and now also in areas of more symbolic calculations such as model-checking, analysis of large graphs — *e.g.* social networks.

The reliability of such programs is too important to be left to chance and good intentions. And those programs will execute on architectures which are expensive and consume a lot of resources: using all these resources to just having a bug is necessarily very frustrating. It therefore seems more appropriate to find errors (*a priori* verification) before the said programs are carried out and whatever the number of processors for a greater portability and maintenance of the proofs of correctness.

The lack of tools is due too the same reasons that parallel programming is hard and to the lack of standards[2] and also to a lack of interest from the scientific computing community for formal methods — which is certainly due to a lack of training. And correctness of parallel programs introduces new difficulties: finding data-race, deadlocks, cyclic communications, *etc.* Parallel libraries are generally well documented but even experienced programmers can misunderstand these APIs, especially when they are informally described.

(b) Some Solutions for the MPI Standard

Some works on the standard MPI exist, a nice introduction could be found in [200]. To summarise, we can find dynamic testing and runtime verification [201–203][3] or debugging solutions [204] (help the checking step-per-step) or model-checking [205, 206] sometimes for only specific properties [207].

A mechanised semantics of C/MPI primitives has been done in [208] (using TLA$^+$ [[53]], the Temporal Logic of Actions) which can be used for the model-checking of MPI programs — and dynamically testing logical assertion. Model-checking of C/MPI programs has also been done in [209] [[54]] [210] [211,212] [[55]] [[56]]. By model-checking the MPI source code and by using an abstraction of the MPI calls (the schemes of communications), an engineer, by push-button, can mainly verify that the program, for example, does not contain any possible deadlocks.

The drawback of these methods is that checking is limited to a predefined number of processors: it is impossible to verify that the program is deadlock free for any number of processors, which is a scaling problem. And increasing this number of processes will increase the verification time (at most exponentially) even if partial order reductions are used to reduce the state space. Most these works are aimed at standard concurrency properties, rather than the functional correctness of the computations carried out by an MPI program. Those tools do no check/prove statically the correctness of the results — only assertion violations. They focus on important properties as deadlocks, resource leaks or data-races. This greatly increases the confidence we can have in the programs but it is not enough if we want to ensure the relevance of the results.

Proving a program for any number of processors brings an additional difficulty: many properties are only prove by induction (over $\mathbf{p}$ the number of processors) that usually trouble automatic provers.

(c) Proposed Solution: Deductive Verification of BSP Programs

Having analysed the problems, in fact, avoiding deadlocks (or data-races) is not sufficient to ensure that the programs will not crash. Mainly, we need to check buffer and integer overflows or liveness. And one can also want a better trust in the code: are results as intended? It is becoming increasingly recognised that "correctness" is not only the safe execution of a program but also formally characterise the intended results and formal properties that hold during the execution. Using a Verification Condition Generator (VCG) tool is the proposed solution. A VCG takes an annotated program as input and produce verification conditions (proof obligations) to provers as output to ensure correctness of the properties given in the annotations[4]. An advantage of this approach is to allow the mixing of the manual proof of properties using proof assistants and automatised checks of simple properties using automatic decision procedures.

But attack frontally all the MPI's API seems too difficult for a VCG tool. Another approach would be to consider well-defined subsets that include interesting structural properties. In fact, many parallel programs are not as unstructured as they appear [9].

Our modest contribution in this subject is to provide a tool for the verification of properties of a special class of parallel programs by providing annotations and generation of proof obligations using a VCG. We

[1] Generally numerical computations but also in the analysis of texts in biology or in social sciences and humanities.

[2] While MPI is now still heavily used; but GPUs and multi-cores paradigms add some new kinds of libraries.

[3] The verification is performed during the execution of the program: this is not a *a priori* verification.

[4] Even if many engineers do not want to write anything else than the programs and thus want push-button tools (such as model-checking) to provide at least safety and liveness, there exists some tools [213] that automatically provide annotations for some properties. But defining the exact meaning of a computation is clearly a human work. We have not yet studied this problem and now we add assertions manually.

choose BSP programs for three main reasons: (1) MPI programs with collective operators can be seen as BSP programs; (2) it is intrinsically a determinist model; (3) as in COQ proofs of BSML programs, the structured nature of BSP programs (sequence of super-steps and clear separation between communications and computations) allows to execute them in a sequential manner. This latter property will be used by our tool to generate sequential programs from BSP ones and by using a traditional VCG as a back-end to generate goals — logical conditions. Also, the structural nature of the BSP programs allow to decompose the programs into sequences of blocks of code, each block mainly corresponding to a super-step.

As stated by J.-C. Filliâtre: *Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it.*[5]

Deductive verification emphasises the use of a logic, to specify and to prove. This discipline is more interested in the verification of small and challenging programs rather than million line long software systems. Deductive verification can be used to verify algorithms as well as programs, if the tool has been designed to verify codes that are executable — in our current work we only provide a tool for algorithm verification. The main concept (known nowadays as Hoare triple [214]) binds together a *precondition* P, a program statement s, and a *postcondition* Q. This triple is usually noted $\{P\}s\{Q\}$. The basic method is the insertion of logical annotations in the programs (called annotated programs) *e.g.* an *invariant* of a loop[6].

From this annotated program (or algorithm), a VCG would produces verification conditions for provers — proof assistants or automatic provers. Basically, the so-called Dijkstra's weakest preconditions calculus (wp calculus) is used. The key advantages of using a VCG are:

- It allows verifying simple properties (as "no overflow") of a program without formally proved its entire correctness;
- Some tools enjoys to automatically insert annotations for some simple properties;
- Using automatic provers enable detecting quickly simple errors.

Writing a proof assistant or a VCG is tremendous amounts of works which should be left to the field experts. The main idea of our work is to simulate the parallelism by using a transformation of the parallel code into a pure sequential one. Therefore, the goal is using a "well defined" verification tool of sequential programs as a back-end for our own verification tool of parallel programs. Furthermore, implementing a VCG for a realistic programming language needs a lot of work: too many constructs require a specific treatment. Reducing the VCG to a core language seems a good approach.

We choose the VCG WHY [215–218] [[57]]. First, it takes as input a small language close to ML avoiding us to handle all the constructs of a real language. Instead, realistic programming languages can be compiled into the WHY input language: WHY currently interprets C programs, JAVA and ADA programs with the help of companion tools [219–221]. Second, WHY is currently interfaced with the main proof assistants (COQ [[58]], PVS [[59]], ISABELLE [[60]], HOL [[61]]) and automatic decision procedures (SIMPLIFY [[62]], ALT-ERGO [[63]], Z3 [[64]], CVC3 [[65]], YICES [[66]], VAMPIRE [[67]]) as back-ends for the proofs obligations. This allows to use these provers for the proof obligations obtained from the parallel programs.

3.1.2 Imperative BSP Programming and MPI's Collective Operators

In this section, we summarises how writing BSP C programs. More details can be find in the APIs of BSPLIB [78], PUB [91] and most BSP libraries. It aims to give an informal semantics to our BSP operations in BSP-WHY. The reader could also easily imagine the JAVA version of these primitives — as in HAMMA [130]. BSP libraries mainly offer operations for both message passing (BSMP) and remote memory access (DRMA)[7]. Some collective communication operations like broadcast are also provided and can easily be simulated by BSMP operations. The C primitives are given in Figure 3.1.

(a) Initialisation and BSP parameters

As in the standard MPI, we first need to initialise our parallel computation which is done using the function bsplib_init. Now, we can query some informations about the machine: **bsp_nprocs** returns the number of processors **p** and **bsp_pid** returns the processor id. To terminate a BSP computation, we use bsplib_done which cleans up all BSP resources.

[5] Although I think it is rather craft than art: one must be more careful and persevering.
[6] A property that is logically valid before the loop and during any step of the loop.
[7] DRMA allows processes to specify shared memories and distant read/write in this memories.

Tools	
void bsplib_init(t_bsplib_params* parameter)	initialises the BSP computation
void bsplib_done()	exits and frees resources
int bsp_nprocs()	returns the number of processors
int bsp_pid()	returns own processor-id
void bsp_sync()	BSP synchronization
BSMP primitives	
void bsp_send(**int** dest,**void*** buffer,**int** s)	Bulk sending of a value
t_bspmsg* **bsp_findmsg**(**int** proc_id,**int** index)	receiving a value
int bsp_nmsgs(**int** proc_id)	number of received messages
void* **bspmsg_data**(t_bspmsg* msg)	returns a pointer to the data of a message
int bspmsg_size(t_bspmsg* msg)	returns the size of a message
DRMA primitives	
void bsp_push_reg(**void*** ident, **int** size)	register a variable for remote access
void bsp_pop_reg(**void*** ident)	delete the registration of a variable
void bsp_get(**int** srcPID,**void*** src,**int** offset,**void*** dest,**int** nbytes)	remote writing to another processor
void bsp_put(**int** destPID,**void*** src,**void*** dest,**int** offset,**int** nbytes)	remote reading from another processor
Subgroup primitives	
void bsp_partition(t_bsp* bsp, t_bsp* sub, **int** nr, **int*** partition)	create a new group of processors
void bsp_done(t_bsp* bsp)	destroys a subgroup

Figure 3.1. The BSP programming C primitives.

(b) Message Passing and Synchronisation

According to the BSP model, all messages are received during the synchronisation barrier and cannot be read before. Barriers are done using **bsp_sync** which blocks the node until all other nodes have called **bsp_sync** and all messages sent to it in the current super-step have been received.

Sending a single message is done using **bsp_send**(dest,buffer,s) where buffer is a pointer to a memory address to send to processor id dest and s is the size in bytes of this block. After calling this routine the buffer may be overwritten or freed.

In the next super-step, each processor can access the received messages (type t_bspmsg). This is done using **bsp_findmsg**(proc_id,index) where proc_id is the id of the source-node and index is the index of the message. To access to the message, we need **bspmsg_data** which returns a pointer to the sending block of data and **bspmsg_size** its size. Also **bsp_nmsgs**(id) returns the number of messages received in the last super-step from node id. Note that the messages of the last super-step are available until the next synchronisation call. At this point the memory used for these messages will be deallocated.

In BSP JAVA libraries, it is the same kind of routines but where serializable objects are communicated.

Remote Memory Access. Another way of communication is through remote memory access: after every processor has registered a variable for direct access, all processors can read or write the value on other processors. Registering a variable or deleting it from global access is done using: **void bsp_push_reg**(ident,size) and **bsp_pop_reg**(ident).

Due to the SPMD (Single Program Multiple Data) structure of BSP programs, if **p** instances share the same name, they will not, in general, have the same physical address. To allow BSP programs to execute correctly, BSP libraries provide a mechanism for relating these various addresses by creating associations called registrations. A registration is created when each process calls **void bsp_push_reg** and, respectively, provides the address and the extent of a local area of memory: registration takes effect at the next barrier synchronisation and newer registrations replace older ones. This scheme does not impose a strict nesting of push-pop pairs. For example :

On processor 0:	On processor 1:
void x[5],y[5];	**void** x[5],y[5];
bsp_push_reg(x,5)	**bsp_push_reg**(y,5)
bsp_push_reg(y,5)	**bsp_push_reg**(x,5)

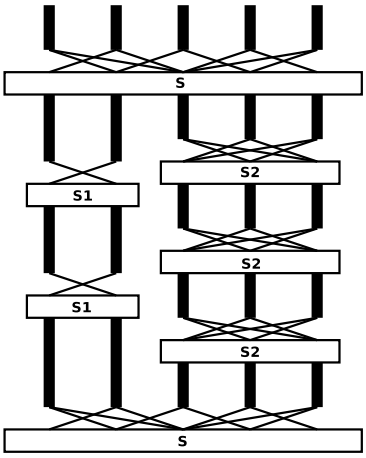
x on processor id 0 would be associated to y of processor *id* 1 — note that this is clearly not a good way of BSP programming. In the same manner, a registration association is destroyed when each process calls **bsp_pop_reg** and provides the address of its local area participating in that registration. A runtime error will be raised if these addresses (*i.e.* one address per process) do not refer to the same registration association. Un-registration takes effect at the next synchronisation barrier. The two DRMA operations are:

1. **bsp_get**(srcPID,src,offset,dest,nbytes), global reading access, copies nbytes bytes from the variable src at offset offset on processor srcPID to the local memory address dest;

2. **bsp_put**(destPID,src,dest,offset,nbytes), global writing access, copies nbytes bytes from local memory src to variable dest at offset offset on node destPID.

All get and put operations are executed during the synchronisation step and all get are served before a put overwrites a value.

(c) Subgroup Synchronisation



A feature (which is not purely BSP) of the PUB is to allow subgroup synchronisation: partition the set of processors into pairwise disjoint subsets; each subset acts like an independent BSP computer. This forces to limit communications and synchronisations primitives to group of processors — as the communicator argument of MPI's collective operations *e.g.* MPI_COMM_WORLD. In left, we show an example of execution of subgroup synchronisation.

Creating a subgroup using bsp_partition is close to MPI's MPI_Comm_dup. The operation takes the current group and create another one using a new number of processors nr and partition is a pointer to node-ids defining the subgroups. Using of subgroups is like a stack, only the lowest subgroup is allowed to generate other subgroups.

All previously defined BSP operations now work for a given group and bsplib_init gives the whole BSP machine at the beginning of the computation. It is to notice that MPI's collective operations also work for a given group of processor. Note that MPI proposes another useful operation which is the MPI_SENDRECV: this routine combines in one call the sending of a message to one destination and the receiving of another message, from another process. This operation can be seen as a BSP's super-step on a subgroup of two nodes. There are three main reasons to want to improve our study to subgroup synchronisation.

First, one of the objectives of this work is to be able, in the short term, to prove MPI programs that are well structured. This can be done by working with MPI programs that only use collective operations and sendrcv [222]. These ensure that the processors will synchronise to exchange their data, in a way not unsimilar to BSP. Collective operations in MPI come with the notion of groups and communicators. It is thus necessary to extend our model to be able to work with programs that synchronise over a restricted group of processors. Second, subgroup synchronisation is part of one of the most used BSP library, the PUB library. In BSP-WHY, we chose to take into account the approaches of both MPI and PUB libraries to subgroup synchronisation. Third, subgroup synchronisation is a solution for programming clusters of multi-cores: one can synchronise only the cores of a node and thus not the whole parallel machine.

3.1.3 Examples of Crashes

Even if the BSP model is theoretically deadlock and livelock free and simplifies the writing of parallel codes, many errors, even deadlocks, can appear in BSP programs in addition of the classical sequential programming errors (buffer and integer overflows, non terminating loops, *etc.*). Take for example the following C code:

```
if (bsp_pid()==0) bsp_sync();
 else asynchronous_computation();
```

here, a deadlock can occur and the parallel machine would crash on some architectures. Communications can also generate errors:

```
int x=bsp_pid();
(* All processors except 0 send a message to 0 *)
if (bsp_pid()!=0) bsp_send(0,(void*)x,sizeof(int));
bsp_sync();
(* processor 0 reads these messages *)
if (bsp_pid()==0)
  for(int i=0;i<=bsp_nprocs()-1;i++)
     x+=(int)bspmsg_data(bsp_findmsg(i,0));
```

processor 0 will read a message (which do not exsit) from itself too. Another example:

```
int x[bsp_nprocs()];
bsp_push_reg((void *)x,bsp_nprocs()*sizeof(int));
bsp_sync();
```

```
(* All processors except 0 write to the x of processor 0 *)
if (bsp_pid()!=0)
 bsp_put(0,(void *)x,(void *)x,bsp_pid()+1,1*sizeof(int));
bsp_sync();
```

the last processor would write over the limits of x of processor 0 and a segfault will occur.

This example is not really an error since it does not crash the machine but gives undeterministic results and thus can disturb the meaning of a program. What happens when there are two distant writings (using the put primitive) of two different processors over the same area of memory? For example:

```
int x[bsp_nprocs()];
bsp_push_reg((void *)x,bsp_nprocs()*sizeof(int));
bsp_sync();
bsp_put(0,(void *)x,(void *)x,0,1*sizeof(int));
bsp_sync();
```

Two solutions are possible. First, forbid this case by adding logical conditions for distant writings. Second, suppose an order of writing of the processors. We have currently chosen the second case since we suppose a deterministic semantics of BSP programs but changing to the first case is trivial to do.

Last example is a lack of composition of the code. Take this simple broadcasting function:

```
void bcast(int proc_src, char *buffer, int size) {
  (* processor src sends its message to every processor *)
  if (bsp_pid()==src) {for(int i=0;i<=bsp_nprocs()-1;i++) bsp_send(i,(void*)buffer,size);}
  bsp_sync();
  (* each processor read the first message from processor src *)
  buffer=(char*)bspmsg_data(bsp_findmsg(src,0)); }
```

wher we suppose here enought allocated memory for each buffer. The problem is that if the function is called in a context where the environment of communications is not empty, then each processor reads an "unpredictable" message. The solution of the PUB is to create a copy of the current group of processors and then performing the exchanges within this new environment of communications. But this requires another synchronisation. Another solution would to specify (formally) that the function could only be used in the context of an empty environment.

Many other errors can be cited: forgetting to register a variable, forgetting a barrier and all errors with pointers of the messages that one can imagine. Proving that programs do not have these incorrect uses of the routines would increase confidence in the codes. This will be even better if you also formally prove the behaviour of BSP programs — at least, the more important parts of the code.

3.2 The BSP-Why tool

From Publications 10

This section is taken from the following publications:

- *in [W4] we describe a part of the* BSP-WHY *tool;*
- *in the ongoing doctoral thesis of Jean Fortin.*

Note that the BSP-WHY tool was written in the OCAML language. Since we chose to keep the syntax of the language very close to WHY, we were able to re-use the open-source code of WHY for a large part of the program. It is the case in particular for the parsing and printing part of the program.

3.2.1 Syntax

The syntax of the BSP-WHY language is the one of WHY[8] with an additional syntax for parallel instructions.

The programming language of WHY (call WHY-ML) is mostly a alias-free ML like language with logical annotations (for pre- post- conditions and loop invariants) inside the brackets and program's *labels* to refer to the values of variables at specific program points. The logic is a specification language (polymorphic multi-sorted first-order logic) which can be used to introduce abstract data types, by declaring new sorts, function symbols, predicates and axioms. Mainly, `ref` e introduces a new reference initialised with e. `while` $e1$ `do` {`invariant` p `variant` t_l} e `done` is a loop of body e, invariant p and which termination

[8] We currently use WHY2 and not the new WHY3, mainly due to a lack of time and also we prefer waiting a more stable version of WHY3.

```
global parameter x: int ref
global parameter z: int ref

let prefixes () =
 let y = ref (bsp_pid void + 1) in
  while (!y < nprocs) do
    bsp_send !y (cast_int !x);
    y ←!y + 1
  done;
  bsp_sync;
  z←x;
  let y = ref 0 in
   while(!y < bsp_pid void) do
     z ←!z + uncast_int (bsp_findmsg !y 0);
     y ←!y + 1
   done
```

```
global parameter x: int ref
global parameter z: int ref

global parameter bsp_sync: unit → {} unit sync {}

logic sigma_prefix : int fparray → int → int

axiom sigma_prefix_def1 : ∀t:int fparray. sigma_prefix(t,−1) = 0
axiom sigma_prefix_def2 : ∀i:int. isproc(i) →
       ∀t:int fparray. sigma_prefix(t,i) = t<i> + sigma_prefix(t,i−1)

let prefixes () = {}
 let y = ref (bsp_pid void + 1) in
  while(!y < nprocs) do
   {
    invariant envCsendIs(j,bsp_pid + 1,y,j,x)
    variant nprocs − y
   }
     bsp_send !y (cast_int !x);
     y ←!y + 1
  done;
  bsp_sync;
  z←x;
  let y = ref 0 in
    while(!y < bsp_pid void) do
     {
      invariant z=x+sigma_prefix(<x>, y)
      variant bsp_pid − y
     }
       z ←!z + uncast_int (bsp_findmsg !y 0);
       y ←!y + 1
   done
   {z=sigma_prefix(<x>, bsp_pid)}
```

Figure 3.2. BSP-WHY code of the direct prefix (left) and with its logical annotations (right).

is ensured by the variant t_l. The `raise` construct is annotated with a type τ. Also, there are two ways to insert proof obligations in programs: "`assert` $\{p\}$; e" which places an assertion p to be checked right before e and "e $\{q\}$" which places a post-condition q to be checked right after e. In annotations, the construct `old`(t_l) denotes the value of term t_l in the precondition state and the construct `at`(t_l, L) denotes the value of the term t_l at the program point L.

Figure 3.2 gives an example of a BSP-WHY program without logical annotations (in left) and the full program expression (in right). It is a simple one-step parallel prefix reduction using BSMP routines: having $\oplus_{k=0}^{\mathbf{i}} v_i$ on each processor where each processor i initially hold v_i (this is the classical MPI_SCAN) for an operation $\oplus$. Here, we used integers and addition for $\oplus$ but a polymorphic program can be considered. The program starts with a distributed parameter x, which contains the initial values, with one value on each processor. The prefixes are computed by the program in the z parameter. We use the user-defined logic term "sigma_prefix(X,n_1,n_2)" to describe the partial sums, that is $\Sigma_{i=n_1}^{n_2} X[i]$.

The program of Figure 3.2 is mainly composed of two `while` loops. In the first loop, each processor sends its value in a message to each processor with a greater *pid* than itself. The instruction `bsp_sync` then executes the synchronisation barrier. In the second loop, each processor computes the sum of all the received values. Note the use of our notations in the program: x designs the value on the current processor, $<x>$ refers to the all the values of x (each processor holds x; it is represented as an array of size $\mathbf{p}$) and x $<i>$ refers to the value of x at processor i. `envCsendIs` is a syntax sugar to describe the communication environment, without having to use the intern list description and its associated functions. t=<<f(pid)>> is another syntactic sugar to ∀i:int. proc(i) → t[i]=f(i).

A special constant *nprocs* (equal to $\mathbf{p}$) and a special variable `bsp_pid` (with range $0, \ldots, \mathbf{p}-1$) were added to WHY expressions. In pure terms (terms without possible side effects), we also have introduced the two special function symbols `bsp_nmsg`(t) and `bsp_findmsg` t_1 t_2: the former corresponds the number of messages received from a processor id t (C function **bsp_nmsgs**(t)) and the latter to get the t_2-th message from processor t_1 (C function **bsp_findmsg**(t1,t2)). The five parallel operations are: (1) `bsp_push` x, registers a variable x for global access; (2) `bsp_pop` x, delete x from global access; (3) `bsp_put` e x y, distant writing of x to y of processor e; (4) `bsp_get` e x y, distant reading from x to y; (5) `bsp_send` e_1 e_2, sending value of e_1 to processor e_2. In order to simplify the presentation, parallel operations (notably DRMA primitives) take simple variables as argument, instead of buffers. In practise, BSP-WHY does manipulate buffers, and adds proof obligations to avoid buffer overflows. The `bsp_send` and `bsp_findmsg` functions can be used to transfer any type of data. For this reason, we use the `cast_int` and `uncast_int` functions, that encapsulates the date in a generic *value* data type. Each primitive of BSP-WHY also works for group of processors; this feature will be discussed later.

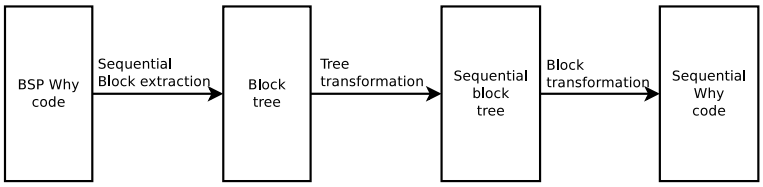


Figure 3.3. Scheme of transformation of BSP-WHY codes into WHY ones.

As in WHY, programs can take parameters that correspond to external values. That allows compositional proof of programs since some parts of the code can be hidden: only their effects on values are needed. The prefix "global" corresponds to a value that would be available on each processor — and possibly different. The WHY language is typed using a simple monomorphic type system for program expressions and with a traditional polymorphic ML type system for purely expressions — logic and terms. This type system is also completed with effects: each expression is given a type together with the sets of possibly accessed or modified variables and the set of possibly raised exceptions. BSP-WHY adds also a special effect to global parameters call "sync" which corresponds to the fact that the code behind the parameter performs at least a global synchronisation — this is obviously the case of the `bsp_sync`. That allows the user to define its own patterns of communications — such as MPI's collective operators or else.

For example, we can note that the WHY tool does not contain any assignment (x `:=` e). This is due to the fact that is can be simulated by the following parameter with effect (defined in a pervasive library):

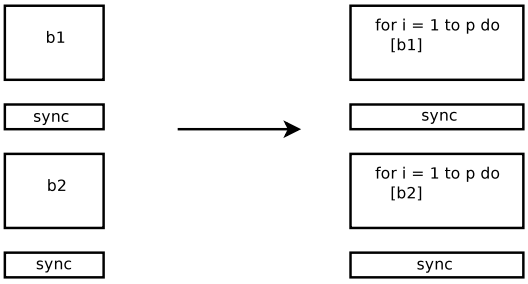
parameter ref_set : x:'a **ref** → v:'a → {} unit writes x { x = v }

In the same manner, the `bsp_sync` operation, barrier of synchronisation, can be simulated by:

global **parameter** bsp_sync: unit → {} unit **sync** {}

(without any communication) that is a specific parameter that holds a `sync` effect. As explain above, that allows users to define their own patterns of communications.

3.2.2 Transforming a BSP-Why-ML Program into a Why-ML one

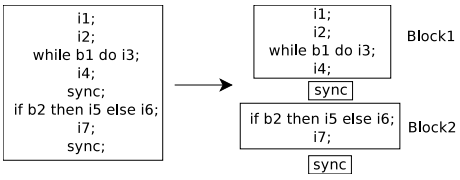


We used a "sequentialisation" of the BSP-WHY-ML codes for their verification, that is we transform BSP-WHY-ML algorithms into WHY-ML ones and thus using the WHY tools to generate adequate conditions for the correctness of the programs. The main idea (in left) is to extract the biggest blocks of code without synchronisation (purely sequential) and them each block is transformed into a *for* loop: for each processor, we executed the block of code.

The main idea of our approach is thus to simulate the execution of a BSP program by a sequential execution which will simulate the entire parallel machine. This way we are able to use the WHY tool — or any tool for checking properties of WHY-ML codes. But in doing so, we need to simulate the memory (environment) of all the computers in a single computer. For this, each variable outside a block (a pure sequential computation) is converted into a **p**-array of values: a different value on each processor. We also need to simulate the functioning of the communication operations.

Let us now detail the different steps of the transformation. The actual transformation of a BSP-WHY program into a WHY one is composed of several steps, which we resumed in Figure 3.3. Let us now detail these different steps of the transformation. The transformation is fully and formally described in the doctoral thesis of Jean Fortin and a part of this transformation was also machine-checked in COQ.

(a) Sequential Block Decomposition



The first step of the transformation is a decomposition of the program into blocks of sequential instructions (see left). The aim is to be able to simulate the execution of a sequential block consecutively for each processor executing it, in a sequential way, instead of the normal parallel execution on each processor at the "same time". In order to obtain the best efficiency, we are trying to isolate the largest blocks of code that are purely sequential.

The result of this phase of the transformation is what we will call from now on a *block tree*, meaning an abstract syntax tree, but where the leaves of the tree are blocks of sequential code, instead of basic expressions. For this, we first must tagging which part of the code are purely sequential or not.

The following instructions potentially influence the parallelism of the program:

- a BSP-WHY parameter defined with the synchronise effect (as the `bsp_sync` instruction);
- a function call, if the function body is determined to have a parallel code — a synchronise effect.

Once we have tagged the syntax tree, the transformation to the block tree is actually almost trivial. It is simply a recursive algorithm on the syntax tree.

(b) Tree Transformation

After having regrouped the sequential parts of the program into blocks, the rest of the tree is just the structure of the parallel mechanisms, and can not be altered. Thus, the transformation on the block tree is made with a traversal of the tree where we apply recursively the transformation. We take a BSP-WHY block tree, and return a WHY block tree, which is structurally identical.

However, several cases need more attention. For example, the `if` and `while` statements needs an additional treatment: when transforming a `if` or `while` structure at the block tree level, there is a risk that a `bsp_sync` instruction might be executed on a processor and not on the other. We generate an assertion to forbid this case, ensuring that the condition associated with the instruction will always be true on every processor at the same time. For instance, if the source code is:

```
while ( (pow_int 2 !i) < bsp_nprocs ) do
 (...)
    bsp_sync void;
 (...)
 i←!i + 1
done
```

the assertion generated would be:

```
assert {∀ proc_i, 0≤ proc_i<bsp_nprocs →
        (((pow_int(2,paccess(i,proc_i))) < bsp_nprocs)
    ←→ (∀ proc_j, 0≤ proc_j<bsp_nprocs →
        (pow_int(2,paccess(i,proc_j))) < bsp_nprocs))};
```

If the condition is true for a processor (`proc_i`), then it must be true for any other processor.

The idea of BSP-WHY is that for a block of instructions that would be run in parallel on each processor simultaneously, we simulate its parallel execution by the use of a "`for` loop" that will run the code sequentially, with one execution for each processor. Because WHY only includes the `while` statement, it actually corresponds to this code:

```
let proc_i = ref 0 in
loopstart:while(!proc_i < bsp_p) do
{
        invariant ...
        variant bsp_p − proc_i
}
e;
proc_i ←(!proc_i) + 1
done
```

When explicitly the loop, one thing is immediately visible. it is necessary to give an invariant to the `while` loop, if we hope to prove anything about the program. Thankfully, the invariant can in general be inferred automatically. The loop consists of the independent execution of the sequential code e, simulated for the processors 0 to $\mathbf{p}$. This means that one iteration of the loop will have executed the code e for one processor. Hence, if we know the post condition *post* that we would like to ensure after the block e, the invariant at the i-th iteration should include: $\forall j \in N, 0 \leq j < i \implies post[j]$.

The meaning is that for all the processors $j < i$, the code e has already been executed, so the post condition *post* at the processor j holds. However, because parallel variables are simulated with the use of arrays in the WHY sequential program, there needs to be another part in the invariant, ensuring that at the ith iteration of the loop, we have not modified the array for the processors i to $p-1$ yet. In other words, for each variable v modified in the loop, we need to have: $\forall j \in N, i \leq j < p \implies v[j] = v[j]@loopstart$.

For finding the invariant of the "for loop", it is relatively easy to find all the variables that are modified by the block, and generate the second part of the invariant for these variables. Guessing an appropriate post-condition for the sequential code is however a bit trickier. In BSP-WHY, we try to guess the most

general post-condition by calling the WHY program on the block of code, and inferring the post-condition. In case the programmer wishes to outline a different invariant, he can do so by giving explicitly a post-condition to the block of sequential code.

For the `bsp_sync` operation (and all synchronous global parameters), it can be useful, in order to prove the correctness of a program, to give a logic assertion just before the `bsp_sync` instruction. This assertion should describe the computations done during the previous super-step and define how are the environments of communications of the processes. In the transformation to the WHY code, this assertion is used in the invariant of the loop executing sequentially the code of each processor. If the logic assertion has not been added, the BSP-WHY tool would search it automatically.

(c) Block Transformation

Finally, we have to give the translation of a single block to the code that can be executed within the "for" loop. The transformation will have the index i of the processor as a parameter, named proc_i. The transformation of control instruction is straightforward, in the same way as previously, by walking the tree recursively. The parallel instructions (put, send, *etc.*) are replaced by calls to the parameters axiomatized in a prelude file. Two difficulties arise: managing variables in expression and in the logic. This first difficulty comes from the fact that a variable x can be translated in different ways depending on its use:

- if the variable is declared locally, and is only used within the sequential block, it is simply translated in a similar variable x;
- if the variable is used outside of the block, it can have different values depending on the processor; if it is not used with a push instruction, it can simply be translated by an array of variables of the same type;
- if the variable is used with a push instruction, it is more difficult to use directly an array, because it is not possible in WHY to transfer pointers to a variable, which would be necessary during the communications; in that case, we chose to use a bigger array, containing all the variables used in DRMA accesses; that way, we can transfer in the communications the index of the variable in the array, rather than the variable itself.

The second difficulty is when translating the logic expressions. It is necessary to translate the variable in the same way as previously. When it is necessary to refer to the variable x as an array <x>, or to the variable on a different processor than the current one, x<i> is transformed in the access to the i-th component of x.

Figure 3.4 gives the final result of this transformation of the prefix examples. The BSP-WHY engine has, as expected, separated the program into two sequential blocks, linked by the synchronisation operation. Around those two blocks, a **while** loop has been constructed, so that the code is executed sequentially for each processor proc_i. We can note that the distributed variables, such as x and z, are translated into arrays of size **p**, using the type $\mathbf{p} - array$. Reading or writing such a variable is done with the `parray_get` and `parray_set` functions, or in the logic world their counterparts `paccess` and `pupdate`.

Local variable, with a lifespan within a sequential block do not need to be translated into an array. For instance, an access to y will remain the same. Note that the WHY source code generated by BSP-WHY is actually not supposed to be manipulated by the end-user, and is in general significantly less readable by a human. It is now possible to use the generated code, and feed it to the WHY program, in order to generate the proof obligations for any supported back-end.

Dealing with Exceptions. We have previously ignored a problem that comes from having exception handling in the language. To better understand the issue, let us consider a simple example program that includes a `try` statement: **try** ((**if** pid=0 **then** raise E **else** void); bsp_sync()) **with** E → void

If we simply follow the transformation as described earlier, the `if` statement would be tagged as not being parallel, so it would constitute a sequential block. But when running this code, the exception `E` will be raised in the first iteration of the "for loop", and will actually stop the loop immediately. It means that the code of the other processors was not executed at all. This means the transformation is not valid for such a program without modification.

To fix both of theses issues, we modified the tagging of parallel code in the first step of the transformation, so that it recognises that a `raise` instruction can have an effect on the parallelism of the program that is if the matching `try` subtree contains some other parallel code.

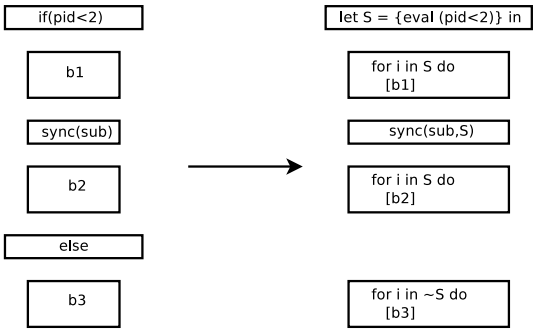
```
parameter x : int fparray ref
parameter z : int fparray ref

let prefixes () = { init_envCsend(envCsend) }
  proc_i ←0 ;
loop0: while (!proc_i < nprocs) do
       {
           invariant (proc_i≥ 0) and
         (∀ proc_j:int. 0≤ proc_j<proc_i →
         (∀ j:int. (proc_j + 1≤ j<nprocs) →
           in_send_n(paccess(envCsend,proc_j),j−(proc_j+1),lcast_int (paccess(x,proc_j)))) and
           nsend(paccess(envCsend,proc_j))=nprocs−(proc_j+1)) and
         (∀ proc_j:int. proc_i≤ proc_j<nprocs →
           paccess(envCsend,proc_j) = paccess(envCsend@loop0,proc_j))
           variant nprocs − proc_i
       }
    let y = ref (bsp_pid (void))+1 in
      while (!y < nprocs) do
       {
         invariant (∀ j:int. (proc_i + 1≤ j<y) →
         in_send_n(paccess(envCsend,proc_i),j−(proc_i+1),lcast_int (paccess(x,proc_i))))
         and nsend(paccess(envCsend,proc_i))=y−(proc_i+1)
           variant nprocs − y
       }
        bsp_send !y (cast_int (parray_get x !proc_i)) ;
        y←!y+1
      done ;
      proc_i←!proc_i + 1
  done;
    bsp_sync (void) ;
    proc_i ←0 ;
```

```
loop1: while (!proc_i < nprocs) do
       {
           invariant (proc_i≥ 0) and
         (∀ proc_j:int. 0≤ proc_j<proc_i→ paccess(z,proc_j)=sigma_prefix(x, proc_j))
           variant nprocs − proc_i
       }
    parray_set z !proc_i (parray_get x !proc_i);
    let y = ref 0 in
      while (!y < bsp_pid void) do
       {
         invariant paccess(z,proc_i)=paccess(x,proc_i)+sigma_prefix(x, y)
           variant proc_i − y
       }
        parray_set z (!proc_i) ((parray_get z !proc_i)+(uncast_int (bsp_findmsg !y 0)));
        y←!y+1
      done ;
      proc_i←!proc_i + 1
  done
    { ∀proc_i:int. isproc(proc_i) → paccess(z,proc_i)=sigma_prefix(x, proc_i)}
```

Figure 3.4. Full prefix example of the generated WHY program.

(d) Dealing with Subgroup Synchronisations

The first step of the transformation, the "decomposition into blocks", remains almost unchanged: tagging of the parallel parts of the code stay identical. The major changes come with the next steps. The feature of subgroup synchronisation needs an extra treatment of the "Tree transformation".



First, the pre-condition of the primitives of synchronisation need to be treated depending of the subgroup: not only all the processors synchronise. Second, the "for loop" would only deal with a subgroup of the processors and we must keep the fact that other processors have no work or a different one depending of the program. The main idea is to generate an appropriate code depending of these subgroups and to check where the flow of instructions depend of these subgroups: where to branch of the **if** statement diverge depending of the subgroups (see left).

It would be senseless to keep guarding the conditional statements as before, since it would only allow the synchronisation of all the processors. Instead, we now dynamically maintain a variable S during the execution of the program, that contains the set of the processors that are running the same branch of the code. To avoid deadlocks, for each bsp_sync(sub,S), we check that all the processors of a subgroup will synchronise at the same time: $assert(\forall i \in S, sub[i] \subset S)$.

It ensures that it is called on a coherent set of processors at any time. For every processor in the set S, which is the set of the processors that will execute the call to bsp_sync, the subgroup that includes the processor is in S. So if one processor in a subgroup calls the synchronisation on that subgroup, every processors in the subgroup are executing it too. This restriction imposed by BSP-WHY is a bit more restrictive than what could be done in, for instance, the PUB library. But if a program can be executed normally, it is generally easy to transform it in an accepted BSP-WHY program. The important property is that inside every subgroups, all the processors do synchronise at the same time.

The main difference in this transformation, compared to the previously defined transformation without subgroup synchronisation, is that the transformation applied to the block tree is now refined with a group of processors. This is useful to handle the **if** statement, as we have seen in the example, and the **while** statement is similar. We also used a parameter that will be used to find what processors have to execute a given code in the case of a **if** or **while** statement.

The idea of the transformation of a local block is similar to the one without subgroup synchronisation. However, instead of executing the block for all the processors successively, we only execute it for the processors that are running that part of the code. The "for loop" means that we execute sequentially the instructions c for all the processors in the subgroup S. That generate a code of the following form:

```
let i = ref 0 in
 while !i<nprocs do
  {
   invariant inv
```

```
    variant nprocs − i
  }
   if proc_in i S then
     c;
   i← !i+1;
done
```

where the proc_in functions just tells if i is in the set S. The determination of the invariant of "for loop" can be made very similarly to what we did before. The difference is that for the processors that are not in S we always need an invariant that their how values have not be modified.

3.2.3 Examples

As test examples, we have written the four following algorithms using BSP-WHY: (1) the simple parallel prefix reduction presented above; (2) a logarithmic version of the prefix reduction (the algorithm combines the values of processors i and $i + 2^n$ at processor $i + 2^n$ for every step n from 0 to $\lceil \log_2 \mathbf{p} \rceil$; (3) a simple parallel Horner method which used a prefix reduction (a sequential Horner method is performed by each processor; subsequently, all the partial results are send to processor 0 which finally evaluate the result applying Horner's rules again); (4) the PSRS in its BSP version [45] — as the one of BSML presented in the previous chapter.

It is easy to see that the number of super-steps is always bounded in the above examples. This is also the case in most BSP programs. Proving terminations is thus generally simple to show.

In this table, we can show how many verification conditions are generated for the above examples. We also show this number when no assertions are given for the correctness of the programs — it is just to have safe execution of the programs without buffer overflow or out-of-bound read of messages. We also show (noted AP) the number of obligations that are automatically discharged by automatic procedures — ALT-ERGO, SIMPLIFY, Z3, YICES and CVC3.

Prog	**Correctness/AP**	**Safety/AP**
Direct Prefix	37/37	19/19
Log Prefix	41/37	21/19
BSP Sort	51/45	27/27
Horner	31/30	17/17

For a simple example such as the direct prefix, all the proof obligations are automatically discharged (proved) by automatic provers. For more complex examples, a few proof obligations are not automatically discharged yet. But safety (no deadlock, no buffer overflow, no out-of-the-bound sending messages *etc.*) is automatically ensured for all examples (except the logarithmic prefix) which is an interesting first result.

Not having all the properties automatically is sad since the generated proof obligations are generally hard to read for WHY and even more for BSP-WHY: this is due to the use of loops over $\mathbf{p}$ for local computations. That also generated harder proof obligations for the provers. For example, logarithm's properties are not currently well interpreted by any automatic provers and thus they fail to prove check bounds accesses in a logarithmic loop.

The key to evaluating the promise of a translation-based technique is in studying the effort needed to prove the generated proof obligations. Currently, many of them are automatically proved and it is thus an encouraging result regarding that it also happens for sequential computations and that our work is to our knowledge the first of its kind. Since these examples are not too difficult, we also believe that by giving more axioms (*e.g.* for log, sqrt, sort, *etc.* which are currently given to the minimum in the WHY libraries) all the proof obligations can be automatically proved.

More experience, or possibly user studies, would be needed to conclude decisively that our assertions are easier to use than existing techniques for specifying that the BSP codes are correct. However, we believe our experience is quite promising. In particular, writing assertions for the full functional correctness of the programs seemed to be quite difficult: we notice that BSP-WHY requires more work from the programmers than a model-checker tool. I think it is the price of Hoare triples and of proving the program whatever the number of processors.

Currently, we have not tested yet correctness of truly subgroup synchronisation algorithms. We are thinking to a n-body computation or to the more complex algorithm of [25] for sparse vector-matrix multiplication. An example would be available in the doctoral thesis of Jean Fortin. In the next chapter, we will give another (and bigger) examples that are parallel state-space computations. In the doctoral thesis of Jean Fortin, we are also working on correctness of the BSP costs of the programs — assuming constant costs for basic operations. For this, we classically adding auxiliary counters to the operations that we want to count, giving sizes of the communications and counting the number of super-steps.

3.3 Mechanised Semantics

From Publications 11

This work (which will appear in its entirety in the doctoral thesis of Jean Fortin) summarises and subsumes the work of [C9], [C8] and [C10].

3.3.1 Background

Given a mechanised operational semantic of a language has many benefits. First, it allows machine-checked compilations, transformations and optimisations of codes. The most important work on this subject is [117]. Second, it gives a formal specification of the language seen as a technical manual as in [223].

Two popular kinds of operational (dynamic) semantics exist: big-step (also known as natural) and small-step — also known a SOS *Structural Operational Semantic*. In a big-step semantic, programs are related to final configurations using a set of (co)-inductive rules whereas in small-step semantic a one-step reduction relation is repeatedly applied to form reduction sequences.

It is generally accepted that big-step semantics present the advantage to be more convenient than small-step ones for proving the correctness (*e.g.* preservation of program behaviours) of program transformations. An (co)-induction on the structure of the big-step evaluation derivations is generally used. On the other hand, small-step semantics are generally preferred, in particular to prove the soundness of type systems or, in concurrent languages, to express the interleaving of processes.

Recognising this fact, we decided to study, using the theorem prover COQ, these two kinds of semantics for BSP (iterative) programs. We developed the two semantics for a core-calculus which is close to BSP-WHY. We can thus focus on the inherent problems of parallel computations. Using a core-language has also two advantages: (1) it is easier to add specific control structures as exceptions handling that cannot appear, for example, in the C language [224]; (2) we have not to manage a complicated memory model.

The big-step semantic (previously studied in [C10]) will be used to mechanically prove the transformation of BSP-WHY programs into WHY ones — this part of the work will appear in the doctoral of Jean Fortin. In [C8], we used a small-step semantic [C9] to mechanically prove a simple optimisation of the code: by transforming classical BSP routines to their high-performance equivalent (the sendings are done asynchronously) when it is possible.

Note that for simplicity of presentation, the semantics rules below do not work for BSP-WHY-ML programs with subgroup synchronisation. They only fit traditional BSP computations. Adaptation of the rules in order to take into account subgroup synchronisation will appear in the thesis of Jean Fortin.

3.3.2 A Big-step Operational Semantic of BSP Programs

A big-step semantic closely models a recursive abstract interpreter. Programs are evaluated (from expressions to values) using inference rules for building finite or infinite (rational) trees.

(a) Definitions

The notions of values and states (also called configuration) are the same as in the semantic of a sequential language (as WHY), with the additional possible value $\texttt{SYNC}(e)$, which describes the awaiting of a synchronisation where e will be performed later. In order to simplify the notation and reduce the number of rules, we use the notation f to design an execution flow, which is either a state of synchronisation or a value. It is used when several rules can be factored.

We note s the environment of a processor. It is a n-uplet which mainly contains the stack of values to be send, the stack of requests of DRMA operations, *etc.* We note $s.\mathcal{X}$ the access to the component $\mathcal{X}$ of the environment s, $\oplus$ the update of a component of an environment without modifying other components and $\in$ the test of the presence of a data in the component.

(b) Local Rules

We first give semantic rules for the local execution of a program, on a processor i. These local reductions rules (*e.g.* one at each processor i) are noted $s, e \Downarrow^i s', v$: e is the program to be executed, v is the value after execution, s is the environment before the execution and s' the environment after the execution.

The semantic of the sequential parts of the BSP-WHY language is not surprising, even though if the semantic contains many rules and many environments (due to the parallel routines), there is no surprise.

$$\frac{}{s, \{\text{p}\}\texttt{bsp\_sync} \Downarrow^i s, \texttt{SYNC(void)}} \qquad \frac{}{s, nprocs \Downarrow^i s, \mathbf{p}} \qquad \frac{}{s, pid \Downarrow^i s, i}$$

$$\frac{s, e \Downarrow^i s', to \qquad 0 \leq to < \mathbf{p} \qquad \{x \mapsto v\} \in s'.\mathcal{E} \qquad s'' = s.\mathcal{C}^{\text{send}} \oplus \{to, v\}}{s, \texttt{bsp\_send}\ x\ e \Downarrow^i s'', \texttt{void}}$$

$$\frac{s, e_1 \Downarrow^i s_1, \texttt{true} \quad s_1, e_2 \Downarrow^i s_2, f_2}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow^i s_2, f_2} \qquad \frac{s, e_1 \Downarrow^i s_1, \texttt{SYNC}(e_1')}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow^i s_1, \texttt{SYNC}(\texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3)}$$

$$\frac{s, e_1 \Downarrow^i s_1, \texttt{true} \qquad s_1, e_2 \Downarrow^i_\infty}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow^i_\infty}$$

Figure 3.5. Examples of "sequential" (local) inference rules of the big-step semantics.

Figure 3.5 gives some examples of local inference rules — executed on a single processor i. Basically, a communication operation adds the corresponding message in the environment. For each control instruction (application, loop, let-in, if-then-else, *etc.*) it is necessary to give several rules, depending on the result of the execution of the different sub-instructions: one when an execution leads to a synchronisation (when the processor finishes a super-step) and one if it returns directly a value. We have thus to memorise as a value the next instructions of the processor. This intermediate local configuration is $\texttt{SYNC}(e)$. In case of infinite computations (noted $\Downarrow^i_\infty$), traditional co-inductive rules can be given as in [225].

(c) Parallel Rules

BSP programs are SPMD ones; consequently, an expression e is started $\mathbf{p}$ times. We model this as a $\mathbf{p}$-vector of e with the environments of execution. A final configuration consists of different values on each processor. We note $\Downarrow$ for this evaluation and the rules are given in Figure 3.6.

First rule gives the base case, when each processor i executes a local (sequential) evaluation $\Downarrow^i$ to a final value. The second rule describes the synchronisation process when all processors execute to a $\texttt{SYNC}(c)$ configuration: the communication are effectively done during the synchronisation phase and the current super-step is finished. The **AllComm** function models the exchanges of messages and thus specifies the order of the messages. It modifies the environment of each processor i. For the sake of brevity, we do not present this function which is a little painful to read and is just a reordering of the $\mathbf{p}$ environments.

The third and fourth co-inductive rules (noted $\Downarrow_\infty$) corresponds to diverging programs: there is at least one processor that diverges or the whole program diverges even if some communications are performed. Note that if at least one processor finishes his execution while others are waiting for a synchronisation, a deadlock will occur and that there is no rule for this case.

By induction and co-induction we have the following results:

Lemma 3

$\Downarrow$ *is confluent.*

Lemma 4

$\Downarrow$ *and* $\Downarrow_\infty$ *are mutually exclusive.*

In [C10], using this kind of rules, we proved that a simple BSP program for the N-body problem is correct and also prove the divergence of a simple BSP program. But using an operational semantic in COQ for proving the correctness of parallel programs (even simple ones) is absolutely not reasonable.

3.3.3 Small-step Operational Semantics of BSP Programs

Small-step semantics specifies the operation of a program, one step at a time. There is a set of rules that is applied continously to configurations until reaching a final configuration if ever. Execution of a program is complete when it reaches the final configuration case or if there exists a reduction step before having this final configuration or the program diverges. If there is no rule, it is a fautly program.

In our parallel case, we will have two kinds of reductions: local ones (on each processor) noted $\overset{i}{\rightharpoonup}$ and global ones (for the whole parallel machine) noted $\rightharpoonup$. For diverging programs, we note these reduction

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, v_i}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow \langle (s'_0, v_0), \ldots, (s'_{\mathbf{p}-1}, v_{\mathbf{p}-1}) \rangle}$$

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \texttt{SYNC}(e'_i) \qquad \mathbf{AllComm}\{\langle (s'_0, e'_0), \ldots, (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1}) \rangle\} \Downarrow \langle (s''_0, v_0), \ldots, (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1}) \rangle}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow \langle (s''_0, v_0), \ldots, (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1}) \rangle}$$

$$\frac{\exists i \quad s_i, e_i \Downarrow^i_\infty}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow_\infty}$$

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \texttt{SYNC}(e'_i) \qquad \mathbf{AllComm}\{\langle (s'_0, e'_0), \ldots, (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1}) \rangle\} \Downarrow_\infty}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow_\infty}$$

Figure 3.6. Global reductions rules of the big-step semantics.

$\overset{\infty}{\Rightarrow}$. Our semantics is thus a set of "rewriting" rules. As we will see, the small-step semantic is harder to define than the big-step one.

(a) Problematic

As for the big-step semantic, most of rules are commons ones. Synchronisation is the only problem. A naive solution is the following global rule:

$$\langle (s_0, \texttt{bsp\_sync}; e_0), \ldots, (s_{\mathbf{p}-1}, \texttt{bsp\_sync}; e_{\mathbf{p}-1}) \rangle \rightharpoonup \langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle$$

that is when all processors are waiting for a global synchronisation and then each processor executes what remains to be done. The problem with this rule is that it cannot evaluate a synchronisation inside a control instruction *e.g.* `if` e_1 `then bsp_sync else` e_3. Different solutions exist:

- Adding specific global rules for the synchronisation inside each control instruction; the drawback is that this implies too much rules;
- Using a global rule with "contexts" (a context is an expression with a hole): the `bsp_sync` instruction replaces the hole within a context on each processor; the drawback is that the use of contexts is not friendly when using a theorem prover as COQ;
- In [226] the authors propose the following rule: $s, \texttt{bsp\_sync} \overset{i}{\rightharpoonup} s, Wait(skip)$ in adjunction with rules to propagate this waiting (as the ones of the big-step semantic) and the following rule

  $$\langle (s_0, Wait(e_0)), \ldots, (s_{\mathbf{p}-1}, Wait(e_{\mathbf{p}-1})) \rangle \rightharpoonup \langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle$$

  but two subtleties persist: (1) the rules add a *skip* instruction that complicates the proofs; (2) in their work, $(e_1; \texttt{bsp\_sync}); e_2$ cannot be evaluate, only $e_1; (\texttt{bsp\_sync}; e_2)$ can.

To remedy to the latter problem, in [C9], we choose to add the congruence (equivalence) $(e_1; \texttt{bsp\_sync}); e_2 \equiv e_1; (\texttt{bsp\_sync}; e_2)$ but that also complicates the proofs. Another solution would consist in having only list of instructions (and not $e_1; e_2$) but that complicates the proofs too.

(b) Local Rules

The solution we propose is the use of a "continuation semantic" described in [227][9]. This semantic mainly allows a uniform representation of configurations that facilitates the design of lemmas.

A configuration is completed with control stack κ. The final configuration is $(s, void \ . \ \epsilon)$, an empty control stack. The control stack represents what has not been executed. There are sequential control operators to handle local control flow. This is close to an abstract machine. In Figure 3.7 we give some examples of local rules. Note that currently, no rule is needed for the `bsp_sync` instruction. The "if" statement is kept in the control stack and, depending on the result of e_1, the control stack gives the evaluation of e_2 with the rest of the stack. A communication primitive consists in simply adding a new value in the environment and in practice all rules of BSP primitives are merged into a generic one.

[9] Using this semantics we also get for free the evaluation of control structures in C (*e.g.* break and continue in loops) if we want to move to a realistic programming language as C.

$$s, nprocs \;.\; \kappa \xrightarrow{i} s, \mathbf{p} \;.\; \kappa$$
$$s, pid \;.\; \kappa \xrightarrow{i} s, i \;.\; \kappa$$
$$s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \;.\; \kappa \xrightarrow{i} s, e_1 \;.\; (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \;.\; \kappa$$
$$s, \texttt{true} \;.\; (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \;.\; \kappa \xrightarrow{i} s, e_2 \;.\; \kappa$$
$$s, \texttt{bsp\_send}\ x\ e \;.\; \kappa \xrightarrow{i} s, e \;.\; \texttt{bsp\_send}\ x\ e \;.\; \kappa$$
$$s, to \;.\; \texttt{bsp\_send}\ x\ e \;.\; \kappa \xrightarrow{i} s', \kappa \quad \text{if } 0 \leq to < \mathbf{p} \text{ and } s' = s.C^{\text{send}} \oplus \{to, v\}$$

Figure 3.7. Examples of "sequential" rules of the continuation semantic.

$$\frac{s_i, e_i \;.\; \kappa_i \xrightarrow{i} s_i', e_i' \;.\; \kappa_i'}{\langle (s_0, e_0 \;.\; \kappa_0), \ldots, (s_i, e_i \;.\; \kappa_i), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1} \;.\; .\kappa_{\mathbf{p}-1}) \rangle \rightharpoonup \langle (s_0, e_0 \;.\; \kappa_0), \ldots, (s_i', e_i' \;.\; \kappa_i'), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle}$$

$$\langle (s_0, \texttt{bsp\_sync} \;.\; \kappa_0), \ldots, (s_{\mathbf{p}-1}, \texttt{bsp\_sync} \;.\; \kappa_{\mathbf{p}-1}) \rangle \rightharpoonup \mathbf{AllComm}\{\langle (s_0, \kappa_0), \ldots, (s_{\mathbf{p}-1}, \kappa_{\mathbf{p}-1}) \rangle\}$$

Figure 3.8. Global rules of the continuation semantic.

(c) Global Rules

As in the big-step semantic, global rules are mainly used to call local ones and **p** configurations have to be reduced. Figure 3.8 gives those rules.

First, the global reduction call a local one. This represents a reduction by a single processor, which thus introduces an interleaving of computations. Communications and BSP synchronisation are done with the second rule: each processor is in the case of a `bsp_sync` with its control stack. The result is only the control stacks since the synchronisation has been performed — with communications.

By induction and co-induction we have the following results:

Lemma 5
$\rightharpoonup$ *and* $\Downarrow$ *(resp.* $\rightharpoonup_\infty$ *and* $\Downarrow_\infty$*) are equivalent.*

Lemma 6
$\rightharpoonup$ *is confluent.*

Lemma 7
$\rightharpoonup$ *and* $\rightharpoonup_\infty$ *are mutually exclusive.*

3.4 Related work

3.4.1 Other Verification Condition Generators

We can cite WHO [228] [[68]] which is a derivation of WHY for ML programming with *polymorphic* functions, and in the same spirit the PANGOLIN system [229]. Both could be interesting for proofs of ML extensions of BSP [R8]. But these tools are not yet stable enough to be used as basis for a parallel extension.

Close to WHY, we can cite the BOOGIE programming language (with the SPEC # annotation language) [230] [[69]]. The YNOT System [231] is an extension to the COQ proof assistant, able to reason about imperative higher-order programs, including functions with side effects as arguments, modular reasoning, while being able to reason about aliasing situations thanks to the separation logic.

3.4.2 Concurrent (Shared Memory) Programs

To our knowledge, the first work on deductive verification of concurrent programs is the famous Owicki-Gries proof system [232]. It is an extension of Hoare logic to parallel programs with shared-variable concurrency. It provides a methodology for breaking down correctness proofs into simpler pieces. First, the sequential *components* of the program are annotated with suitable assertions. Then, the proof reduces

to showing that the annotation of each component is correct, and that each assertion of an annotation is invariant under the execution of the actions of the other components — the so-called *interference-freedom* of proof outlines. The main drawback of the Owicki-Gries method is that it is not compositional: to perform the interference-freedom tests for some component once requires information about the implementation of all other components. Another drawback is the massive use of *ghost* codes (auxiliary variables) needed for proving parallel programs using the Owicki-Gries method. For example, the number of interference-freedom tests is polynomial to the number of sequential components. Note that a formal analysis of what is provable without these ghost codes has been done in [233].

In order to remedy to the above problems, different solutions have been proposed. In [234, 235], the authors proposed a compositional extension of the Owicki-Gries's rules, a wp-calculus, each implemented in the theorem prover ISABELLE. The authors also applied their method for verifying a concurrent garbage collector [236]. Three advantages of the method are (1) the use of a theorem prover ensures a great confidence in the implementation; (2) the method allows verification of open systems, *i.e.* systems which interaction with the environment can be specified without knowing the precise implementation of the environment; this makes the method suitable for top-down design; (3) parametrised concurrent programs (programs with an unknown and unbound number of threads) can be directly verified in the system; this is achieved by modelling the parallel constructor such that its argument is a list of component programs; then, the length of the list can be fixed or left as a parameter. The main drawback (except that it only works for shared-memory programs) is still the number of generated conditions, that rely on "rely-guarantee" which seems to be not provable by automatic provers: they require extensive human guidance.

Notice also the work of [237] where annotations are used to prove safety of concurrent ADA programs, that is, they are deadlock free and without failure. Now, there are also some studies of proof obligations for concurrent programs; for example [238] presented a *Concurrent Separation Logic* as an extension of Separation Logic for reasoning about shared-memory concurrent programs with Dijkstra semaphores. [239] presents an operational semantics for Concurrent C minor which preserves as much sequentially as possible (coarse-grained spirit), by talking about permissions within a single thread instead of concurrency. This semantics is based on ideas from Concurrent Separation Logic: the resource invariant of each lock is an explicit part of the operational model. This model is well suited for correctness proofs done interactively in a proof assistant, or safety proofs done automatically by a shape analysis such as [240]. However, currently no tools for generating proof obligations are provided and it is not clear how hard the obligations would be. A first work in this direction has been done in [241].

[242] presents a type system that prevents data races and deadlocks (as in [239]) by enforcing that locks are acquired in a given locking order, and this order can be changed dynamically. This system supports thread-local objects and coarse-grained locking of shared objects. The type system allows concurrent reading only for immutable objects.

In the same way, [243] presents a sound and modular verification methodology (implemented for an experimental language with some not-trivial examples) that can handle advanced concurrency patterns in multi-threaded, object-based programs. It prescribes the generation of verification conditions in first-order logic (well-suited for solvers). The language supports concepts such as multi-object monitor invariants, thread-local and shared objects, thread pre- and post-conditions, and deadlock prevention with a dynamically changeable locking order. [244] extends the BOOGIE framework for concurrent programs with a kind of locking strategy. In the same way, a VCG for concurrent C programs has been designed in [245] [[70]]. This VCG has been used for the development of the Microsoft Hyper-V hypervisor — see Chapter 5 for a discussion on hypervisors. The hypervisor is highly optimised for multi-core hardware and thus contains a number of custom concurrency control mechanisms and algorithms, mostly using fine-grained concurrency control. The model, with uniform treatment of objects and threads, is very similar to the one employed in Concurrent SPEC # and CHALICE [246]. These tools are well defined for share-memory concurrency but not for distributed and HPC computing.

It is not clear if *locking strategies* are very suitable for high-performance applications [7]. BSP is by nature a coarse-grained and deadlock-free model which is used for high-performance problems and now in multi-core/GPU applications. Even if proof of concurrent programs is clearly useful (servers, *etc.*), parallel programming is not concurrent programming. High-performance programs are much simpler [7] (many time more coarse-grained) and BSP programs are even simpler. They can clearly be simulated by share-memory fork/lock concurrent programs by explicitly separating the local memories and allowing communications with copies of the data [45]. Global synchronisation would be implemented using a sufficient number of locks. But, that would not use the structural nature of the BSP programs and the understanding of the program to simplify the obligations.

3.4.3 Distributed and MPI Programs

A first work on distributed programs was done in [247]. It has served as a basis for the TLA$^+$ logic. But no work or a tool for VCG calculus was proposed.

In [248], the author proposes a data-parallel (SIMD, Single Instruction Multi Data) extension of C and its embedding into a verification environment based on ISABELLE. The data-parallel operations are mostly binary like operations on arrays. Another work on data-parallel programs is the one of [249]. The authors have designed a set of Hoare's rules and a wp-calculus for proving data-parallel programs. Exchange of data is done using shared arrays. The two drawbacks are (1) there is a lack of mechanised proofs; (2) the programs need a lot of annotations (not only loop invariants) which can repel a user; (3) the method does not hold if a program modifies the variables in the assertions (a data-race). For the latter, auxiliary variables can be used but make the proofs slightly more complex.

MPI is the most used library for high-performance computing. It is therefore natural to study safety issues related to MPI programs. But this is very challenging due to the number of routines (more than one hundred), the concurrent nature of these primitives, the lack of formal specifications — even if works such as [250,251] exist, some cases are not taken into account because of the lack of specification and too much dependence on the architecture. This enormous number of routines in the API makes difficult to trust any formal specification of a complete MPI*e.g.*, a non-trivial case could have been forgotten?

They are many works and tools dedicated to MPI. Surveys could be found in [200, 252–257] and in HPCBUGDATABASE [[71]]. These tools help to find some classical errors, but not all of them. For example, in [258], the authors propose to check dynamically if the parameters of collective operators are consistent — *e.g.* if the size of data to send is non-negative. Note that this kind of tools works well for many situations in development phases, but is not sufficient.

[211] presents a tool that directly *model-check* the MPI source code, executing its interleaving with the help of a verification scheduler (producing error traces of deadlocks and assertion violations). It allows an engineer to check its MPI code against deadlocks on a **p**-processors machine before running it on this machine. A large number of MPI routines (including DRMA ones [259]) are considered. This technique was also used to find *irrelevant barriers* [260]. The main idea (which is also the one of [210,261]) is to analyse the code by *abstract interpretation* and to produce an output for model-checkers by eliminating all parts of the code which are not in the *communication schemes*. First implementations used the model-checker SPIN, but now specific and more efficient checkers are considered. The main advantage of these methods is to be "push-button". The two main drawbacks are (1) they only consider deadlocks and assertion violations (it is still an important problem for MPI programs) (2) programs are model-checked for a *predefined* number of processors which is less than 64 in [211]. That is clearly not a *scalable* approach. The BSP spirit is to write programs for any number of processors, so proofs of BSP programs should do the same thing. A last problem with model checking parallel programs is state explosion, *i.e.* the fact that the number of states of a program typically can grow dramatically with the number of processes.

As said above, these methods generally cannot scale beyond a relatively small number of processes, but defects, which usually appear only in large configurations, can often be detected in much smaller configurations using symbolic execution. This original solution is the TASS [262] [[72]] and FEVS tools [263] [[73]]. TASS uses *symbolic execution* and *explicit state enumeration* techniques to verify that safety properties of MPI programs hold for all possible executions within user-specified bounds. TASS has its own internal support for simplifying symbolic expressions, placing them into a canonical form, and dispatching some of the proof obligations; for those it cannot dispatch itself, it uses CVC3. TASS can also use comparative symbolic execution (in the spirit of FEVS) to verify that two programs are functionally equivalent (*i.e.* "input-output equivalent"). The basic idea is to construct a model in which the two programs run sequentially, one after the other, and at the end the outputs are compared. The state space of this model is then exhaustively explored. This technique is particularly useful in computational science, to compare a complex parallel version of a program (the "implementation") to a simple, trusted sequential version (the "specification"). Techniques to verify program assertions using symbolic execution exhibit a significant limitation: they typically require to impose (small) bounds be imposed on the number of loop iterations. The use of *loop invariants* allows to overcome this limitation. In [264], the author proposes a solution using a new symbolic execution technique (with special "collective loop invariants") that he uses to verify assertions in MPI programs with *unbounded loops*. Programs are then checked (a model-checking technique using POR reductions) for unknown sizes of data — but still for a fixed number of processor only. But the author note that discovering these collective loop invariants is currently to the charge of the programmer (as in our work) and these invariants are limited; for example, there is no way to express that the number of messages is invariant. The method still has the advantage of greatly reducing the number of necessary loop invariants.

The approaches of symbolic verification as well as VCG tools, suffer to the main limitation: as it now stands, models of the programs must be built by hand. This requires significant effort and a degree of skill from the user. The ideal situation would be to have tools that automatically extract the models from source code, at least for specific domains [265, 266].

Currently, we are either not aware of verification condition generators tools for MPI programs. We think that performing a sequential simulation (as done by our BSP-WHY tool) of any kind of MPI programs is not reasonable. Continuations would be necessary to simulate the interleaving of processes: that would generate unintelligible assertions. But collective MPI routines, which simplifies the life of parallel programming [267], can be seen as BSP programs, and certainly many MPI programs could be transformed into BSP ones. Automatically translating this class of programs is a possible way to analyse MPI programs [125]. We leave the aim of substantiating this claim for future work.

The only exception is the work of [268, 269] with their prototype tool call HEAP-HOP [[74]]. HEAP-HOP is a VCG for programs with synchronous and copyless sending. A concurrent separation logic is used for the assertions (pre- and post-conditions and loop invariants). As an example, the authors propose to verify a load balancing algorithm for binary trees for two processes. This tool is thus able to take into account some MPI programs. But it is still limited to a fixed number of processes – *e.g.* two for the load balancing algorithm of a consumer/producer algorithm. Using BSP-WHY, the user can prove its programs for any number of processors.

3.4.4 Semantics and Proof of BSP Programs

To our knowledge, the first work on a formal operational semantic of BSP is [270]: the author gives a small-step semantic using its own primitives of its own core language. Neither mechanised work nor applications have been done. The interests and examples of the use of mechanised semantics for certified program verifiers are given in [271]. In [272], the author gives a mechanised proof of the results of the weakest preconditions calculus used in WHY. A mechanised big-step semantic of WHY were given. The author used massively dependent types whereas we choose a simple model of the language in the spirit of [117]. But our work on the proof of the transformation (of BSP-WHY codes to WHY ones) and the results of [272] can clearly be associated to gain more confidence in the outputs of BSP-WHY.

Different approaches for proofs of BSP programs have been studied. In [R8] (see Section 2.5), functional BSP programs have been proved correct. In [C10], we presented the correctness of a classical numerical computation (the N-body problem) using a mechanised operational semantics. But, by using semantics inside COQ, proofs of correctness were too hard.

The derivation of iterative BSP programs using the Hoare's axiom semantics [273] followed by the generation of correct C code [274] also exists. The two main drawbacks of this approach is a lack of an implementation of a dedicated tool for the logical derivation, which implies a lack of safety; users make hand proofs which are not machine checked; moreover, it is impossible to verified users existing codes.

The derivation of iterative BSP programs using the Hoare's axiom semantics has also been studied in [275–278]. More recently, these works were extended for subgroup synchronisation in [279]. All of these approaches lack of mechanised proofs. Moreover, they are close to refinement "à la B" since they give logical rules (close to Hoare logic's axioms and inference rules) for derived algorithms from specifications — a wp calculus is also given in [279]. On the contrary, using deductive verification, we begin with a program and by adding logical assertions, we prove the correctness of the said program.

A work on proving determinism, using assertions in the code, of multi-threaded JAVA programs with barriers can be found in [280]. The authors note that *there seemed to be no obvious simpler, traditional assertions that would aid in catching non-deterministic parallelism.* In our case of BSP programs, this work is simple — but still limited to BSP programs.

Another work on concurrent threading with barriers is [281]. The authors have developed and proved sound a concurrent separation logic for barriers of threads. An interesting point is that the proofs are machine-checked in COQ. The authors also showcase a program verification toolset that automatically applies the logic rules (Hoare logic) and discharges the associated proof obligations. It is thus a work for derivation of formal specification into correct parallel programs. The drawback (as in [268] and partly in [235]) is that only programs with a predefined constant number of threads (*e.g.* two for a producer-consumer problem) can be considered. For HPC, we prefer to have correct programs for an unknown number of processors in a data-parallel fashion. Another interesting work is [282] in which, using the theorem prover COQ, the authors give a mechanised deductive verification of shared-memory concurrent algorithms for software barriers (of synchronisation) of multi-threaded programs. But assertions are hard to understand and especially the number of threads is still bounded.

3.5 Lessons Learnt From this Work

First, creating your own tool for deductive verification needs a great carefulness and having machine-checked proofs (or at least a part) is not only a way to do well in the academic: it is during the design of the mechanised semantics that we highlighted the problem of the exceptions. By thus formalising our semantics and a part of the methods of BSP-WHY in COQ we have gained a full understanding of the difficulty involved in designing correct proof methods for the verification of parallel programs. The level of detail required for such a formal work naturally leads to approaching each step of the formalisation with a critical eye, considering first studying alternatives that could simplify the formalisation. For example, our mechanised semantics do not use at all dependent types (in contrary of [272]) for the simple reason that we do not manage them in our proofs. Our semantics are thus close (in their design) to the ones of [117]: we want to extend this work for BSP programs so it seems natural to do the same kind of semantics. However, to understand the theorem proving techniques, that involved in many formalisations, requires a great deal of time, effort and mistakes.

Second, the interesting part of the verification process is to finally understand the program and find an intuitive proof of their correctness. However, the projection of this intuition into assertions implies, in general, changing and tuning the assertions too many times and a great deal of effort is expended to get the details right. For example, the generated conditions of BSP-WHY are really unreadable and only automatic provers could manage them. But these SMT provers regularly give strange results. Sometimes they found the result and sometimes not without any comprehensible reason. Sometimes they failed due to the add of a new axiom that has nothing to do with the current job. This is thus very tiring when experiments are performed. The tool presented here does not directly help in finding the right annotations but at least automates the iterative process of changing assertions and checking the proof again. From our experience, we do not recommend trusting paper and pencil proofs of correctness of parallel algorithms and programs. Especially if the invariants of the loops are implicitly given — what is traditionally done in the field of parallel model-checking as we will see in the next chapter.

Finally, our approach does have several limitations. First, as it now stands, annotating sequential and parallel programs has to done by hand. This requires significant effort and a degree of skill of the user. The ideal situation would be to have tools that automatically extract some properties (the full specification is a dream), and indeed a great deal of research on this subject has been carried out, at least for some specific domains such as numerical computations [265, 266].

4 BSP Algorithms for the Verification of Security Protocols

*This chapter summarises the work of [C1], [C2], [C3], [C6] and [W2]. This work was done within the framework of the doctotal thesis of Michael Guedj (defence in October 2012) co-directed with Pr. Franck Pommereau (*IBISC *Evry) with a funding from Digiteo ("Région Paris"). This work is also part of the project* SPREADS *(*ANR *funding).*

We reminder that, in spite of their apparent simplicity, secure protocols are notoriously error-prone [10]. Formal methods are needed for their analysis. One of the formal methods is model-checking: the execution of a protocol is simulated, making it easier to verify certain security properties. Model-checking is well-adapted to find flaws (but is generally limited to a bound number of agents) while dedicated tools are better to finally prove the correction — but these tools can find false flaws or cannot accept the protocol due to their how limitations.

In what follow, we design, benchmark and mechanised prove correct (using the tool of the previous chapter) some BSP algorithms for the parallel state-space construction of security protocols. We also extend and benchmark those algorithms for the verification of logical temporal formulas.

Contents

4.1 Why Finding (Potential) Flaws of Security Protocols

4.1.1 Vocabulary of Security Protocols

Security protocols specify an exchange of *cryptographic messages* between *principals*, *i.e.* the agents (*e.g.* users, hosts, or processes) participating in the protocol. Messages are sent over open *networks*, such as the Internet, that cannot be considered secure. As a consequence, protocols should be designed to work fine even if messages may be eavesdropped or tampered with by an *intruder* — *e.g.* a dishonest or careless agent. That aims at providing security guarantees such as authentication of principals or secrecy of some piece of information through the application of cryptographic primitives.

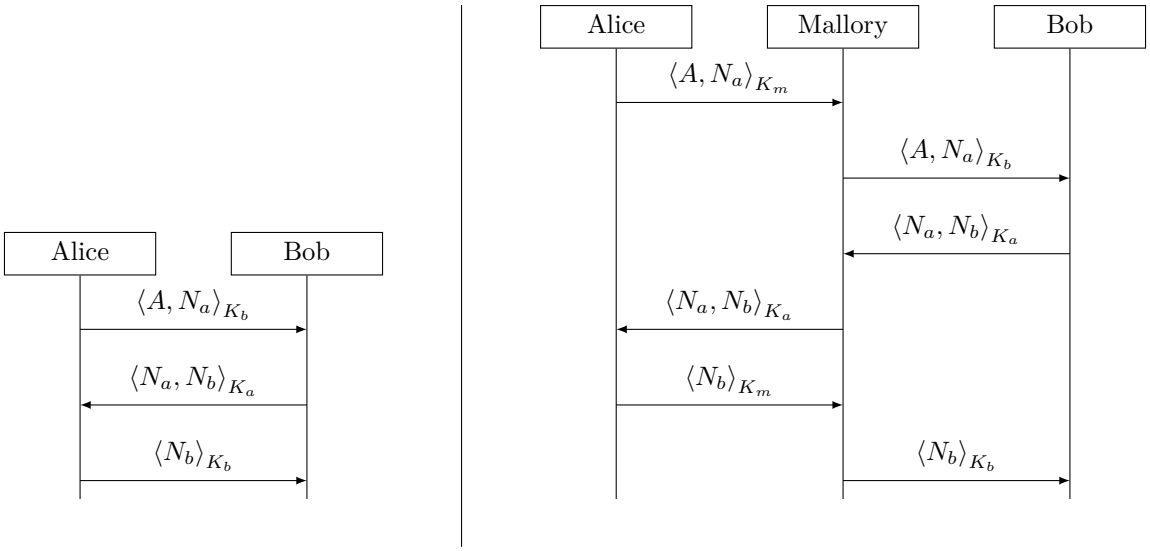


Figure 4.1. An informal specification of the NS protocol (left) and one attack (right) where Mallory (intruder) authenticates as Alice with Bob.

We suppose the use of keys sufficiently long of the best-known cryptographic algorithms to prevent a brute force attack in a reasonable time. This is the well-known *perfect cryptography*. The idea is that an encrypted message can be decrypted only by using the appropriate decryption key, *i.e.* it is possible to retrieve M from the message encrypted M_K only by using K^{-1} as decryption key.

In the following, we will use the following elementary vocabulary on security protocols: *key* (public or not), a value that can crypt or decrypt a message; *fresh term*, *e.g.* nonce, a new value each time the protocol is used (random numbers); *sessions*, Each use of the protocol is call a session (an agent can execute more than one time the protocol); *agents*, the participants of the protocols including intruders. Figure 4.1 (in left) illustrates the NS protocol which involves two agents Alice (A) and Bob (B) who want to mutually authenticate — N_a and N_b are nonces and K_a, K_b are the public keys of respectively Alice and Bob.

Agents perform "ping-pong" exchange of data exchange and what kind of attacks do there exist against security properties of protocols? We can find the following typical actions of the intruder (not all are listed): *interruption*, the communications are destroyed or becomes unavailable or unusable; *eavesdropping*, an unauthorised party gains access to the communication; *modification*, an unauthorised party not only gains access to but tampers with the network; *fabrication*, the intruder inserts counterfeit data into the network; *traffic analysis*, an unauthorised party intercepts and examines the messages flowing over the network in order to deduce information from the message patterns; *etc.*

Some well-known strategies that an intruder might employ are: *man-in-the-middle*, the intruder imposing himself between the communications between the sender and receiver; *replay*, the intruder monitors a run of the protocol and at some later time replays one or more of the messages; *etc.* We can enumerate the following (most commonly and important considered) security properties: *authentication*, *confidentiality*, *integrity*, *availability*, *non-repudiation* (for commercial protocol with a contract [284]), *anonymity*, *eligibility*, *fairness*, *public verifiability* — for an voting protocol [285–287].

All these properties and attacks are well explain in [10]. In Figure 4.1 (right), we show the well known *flawed* (due to a "man-in-the-middle" *logical attack*) of the NS protocol when initiated with a malicious third party Mallory (M); it involves two parallel sessions, with M participating in both of them.

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 [12]. 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 intruder. Therefore, there is no need to assume the existence of multiple attackers or dishonest agents, because they together do not have more abilities than the single Dolev/Yao intruder.

4.1.2 Model-Checking Security Protocols

Formally verifying security protocols is nowadays an old subject but it is still relevant. Different approaches exist as model-checking [17, 76, 288], annotations in a process algebra [19], typing [289, 290] and

theorem proving [291,292]. Some tools are also dedicated to this work as Strand Spaces [293], AVISPA [294] [[75]], SCYTHER [295] [[76]], TAMARIN [[77]], PROVERIF [296] [[78]], NRL [297] [[79]], *etc.* Some of them generate certificates for a mechanised verification using a theorem prover [292,298–301]. Most of the dedicated tools can only prove specific properties (mainly secret) of protocols, for an unbound number of agents; but they are also limited to those properties — *e.g.* fairness is not taken into account.

By focusing on verification of a *bounded number of sessions*, model-checking a protocol can be done by simply *enumerating* and *exploring all traces* of the execution of the protocol and looking for a violation of some of the requirements. Verification through *model checking* consists in defining a formal model of the system to be analysed and then using automated tools to check whether the expected properties (generally expressed in a temporal logic) are met or not.

(a) Advantages and Disadvantages of this Method

Model-checking attempts to find a reachable state or trace where some supposedly property fails – *e.g.* secret term is learnt by the intruder or an incorrect authentication occurs. To ensure termination, these tools usually bound the maximum number of runs of the protocol that can be involved in an attack.

The advantage of this method is to be *automatic* and is *not limited by the protocols themselves.* For example, PROVERIF used approximations of the protocols (formalise in a π-calculus) but sometime finds false attacks in case of repetition of actions. The limitation of the actions of agents (*e.g.* keep data-structures as a list of trusts servers [302]) is also limited to the expressiveness of the input of the model-checker — which is generally more expressive than dedicated tools. Furthermore, model-checking allows construction of a trace of execution of the protocols (in case of finding a flaw) that most dedicated tools cannot. This trace may come from a complex logical property expected from the protocols that dedicated tools can not provide — *e.g.* PROVERIF is mainly limited to secrecy and authenticity and cannot infer fairness. Moreover, we can benefit from classical reduction techniques of model-checker algorithms as BDD s, bit-hashing, partial orders, symmetries, distributed computations *etc.* The main drawback is that model-checking is generally limited to find flaws and not to prove a protocol is correct. Therefore, they can only detect attacks that involve no more runs of the protocol than the stated maximum.

To model-check a security protocol, one must first construct its state-space (all executions of the protocol) and then check properties usually expressed in a temporal logic. We now formally define them.

(b) State-space Construction

The state-space of a protocol is thus all the executions of the agents of the protocol considering all the messages built by the intruder. The state space construction consists in constructing a LTS.

Definition 1 (*Labelled Transition System (LTS)*).
It is a tuple (S, T, L) where S is the set of states, $T \subseteq S \times S$ 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 $\mathrm{LTS}(s_0, \mathsf{succ})$, 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.

Now supposing a set $\mathcal{A}$ of atomic propositions, we have:

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

Mainly a Kripke structure is a LTS adjoining with a labelling function which give verity to given state.

Definition 3 (*Path and related notions*).
Let $M \stackrel{\text{df}}{=} (S, T, L)$ be a Kripke structure.

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

State-space construction may be very consuming both in terms of memory and execution time: this is the so-called *state explosion problem.* For reducing the state-space, different "generic" solutions exist as

partial order (POR) [303], BDD [304], symmetries [305], unfolding [306], *etc.* Another solution, which is the subject of this chapter but does *not* reduce the state-space, consists in exploiting the larger memory space available in distributed systems [307,308]. Parallelling 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. In this chapter, we will present BSP algorithms for the construction of state-space of security protocols.

Building the state-space is also the first step for the verification (model-checking) of properties of the protocols. And, in general, one may identify two basic approaches to model-checking. The first one uses a global analysis to determine if a system satisfies or not a formula; the entire state-space of the system is constructed and subjected to analysis. However, these algorithms may be used 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 analysed in order to determine whether a system satisfies a formula or not. 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.

(c) Formal Representation of Security Protocols

In this chapter, we model security protocols as a LTS and we consider a Dolev-Yao attacker that resides on the network. 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.

As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* called ABCD [309, Sec. 3.3] (not presented in this document) allowing easy and structured modelling. This algebra contains control flow that embeds a programming language (PYTHON) as colour domain. This algebra is part of the SNAKES library [309] which is a general Petri net library that allows to model and execute PYTHON-coloured Petri nets: tokens are PYTHON objects and net inscriptions are PYTHON expressions.

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. Messages can be modelled by tuples and cryptography can be treated symbolically, *i.e.*by writing terms instead of 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.

Agents' identities are just positive integers because no such values are 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, a PYTHON class **Nonce** is used. 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.

Cryptography is considered to be perfect, so the intruder 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 this aim, the Dolev-Yao inference rules (not shown here) have been encoded in Python and the intruder maintains a knowledge base $\mathcal{K}$ containing all the information learnt so far and repeatedly executes the inference rules over the knowledge[1]. The knowledge part of the attacker is modelled by a Petri net place *i.e.*, by an ABCD buffer. Notice also that, when composing a new message, the intruder may rebuild the message it has just stolen. This corresponds to a message eavesdropping, which the attacker is also allowed to do. We refer to [74] for more details and examples of security protocols formally model using ABCD.

[1]To also reduce the combinatorial explosion in the state-spaces, an instance of the Spy is given as 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. 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. So, the attacker uses the protocol signature to restrict the knowledge to valid messages or message parts.

(d) Temporal Logic and their Verification

Considerable attention has been devoted to the development of automatic techniques (model-checking) procedures, for verifying finite-state systems against specifications expressed using various *temporal logic* as CTL* which subsumes the two useful logic in verification that are LTL (linear-time temporal logic) and CTL (Computational tree logic). Temporal logic are popular property-description languages since they can describe event orderings without having to introduce time explicitly. There are two main kinds of temporal logic: linear and branching. In LTL, each moment in time has a unique possible future, while in CTL, each moment in time has several possible futures.

A temporal logic is used to reasoning about propositions qualified in terms of time. Temporal logic 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".

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 logic are more expressive than CTL*. It is the case of the μ-calculus and game-semantic logic as ATL* [310] which are not study in this document.

4.1.3 Why Verifying Model-checkers

In the following, we will model-check security protocols against temporal logical formula that will express security properties. Before presenting our BSP algorithms for this verification and describe how we can proved (mechanised) the correction these algorithms, we give here some arguments about the value of the (mechanised) proof of correction of algorithms and programs for model-checking.

(a) Generalities

Model checkers (MCs for short) are often used to verify safety critical systems. The correctness of their answers are thus vital: many MCs can generate a counterexample computation if a property of the model fails; on the other hand, MCs produce only the answer "yes" which forces, in the two cases, to assume that the algorithm and its implementation are correct.

On the one hand, we would like verification tools to feature powerful automation, but on the other hand, we also want to be able to trust the results with a high degree of confidence. And MCs, like any complex software are subject to bugs and if MCs are widely used tools, there exist surprisingly few attempts to prove them. We see three main reasons: (1) they involve complicated computational logic, algorithms and sophisticated state reduction techniques; (2) because efficiency is essential, MCs are often highly optimised, which implies that they may not be designed for rigorous system maintenance; (3) these techniques are developing rapidly and thus MCs are often updated. But there is a more and more pressing need from the industrial community, as well as from national authorities, to get not just a boolean answer, but also a formal proof, which could be checked by an established tool such as COQ or ISABELLE. This is required in Common Criteria certification of computer products at the highest assurance level EAL 7 [[80]].

(b) Main Problems of Certifying Model-checkers

For this purpose, different solutions have been proposed and Figure 4.2 summaries the following methods. The first one is to prove the MC inside a theorem prover [311] and use the extraction facilities to have a pure functional machine-checked program — as the compiler of [117]. In this manner, a μ-calculus computation has been done using COQ in [312]. The second and more common approach, in the spirit of Proof-Carrying Code [313] (PCC for short; in some article, authors speak of producing an "invariant" which is a close work), is to generate a "certificate" (also called an evidence) during the execution of the MC that can be checked later or on-the-fly by a dedicated tool or a theorem prover. This is the so-called "certifying model-checking" [314] which is an established research field [315, 316]. In this way, users can re-execute the certificate/trace and have some safety guarantees. This approach has the advantage that even if the MC is buggy, its results are checked by a trusty tool.

But, any explicit MC may enumerate state-spaces of astronomical sizes (the famous state-space explosion problem), and mimicking this enumeration with proof rules inside any theorem prover would be foolish even if specific techniques [317] and optimisations of the abstract machine of theorem provers [318] are used: this problem does not arise when finding a refutation of the logical formula (the trace is

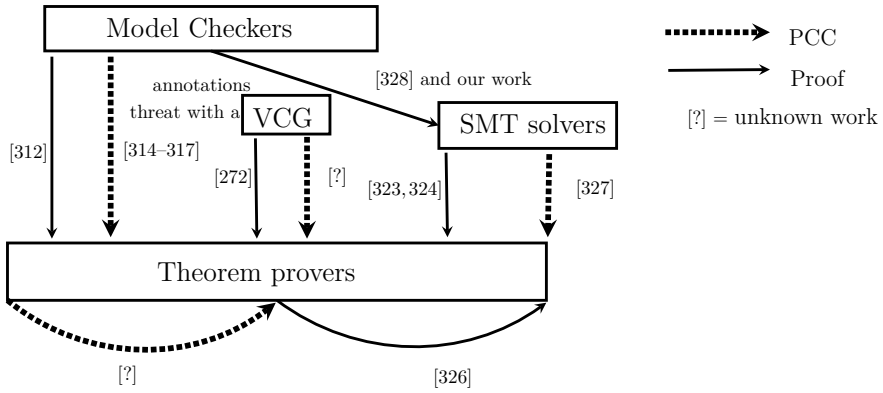


Figure 4.2. Different ways for proving model-checking algorithms.

generally short) but when the answer is "yes" since the entire explicit state-space (or at least a symbolic representation) needs to verify the checked properties. In this way, proof generation could hamstring both the functionality and the efficiency of the automation that can be built into these tools. This performance issue of an "inadequate use" of PCC is discussed in [319, 320]: PCC only focuses on the generation of independently-checkable evidences that the compiled code satisfies a simple behavioural specification such as memory safety; the evidence can then be checked efficiently. Using PCC (or even Foundational PCC) for state-space is the same as computing it a second time. Note that this problem also arises for the extracted functional program of [312]: it would be too slow for bigger models. In fact, the drawback of proof certificates is that verification tools have to be instrumented to generate them, and fully expanded proofs can have sizes that are exponential in the size of the conjecture.

All this finding was also made in [321] and [322] where the authors present a COQ's development (and benchmarks) of binary decision diagrams (BDD s) and tree automata. They conclude that PCC s are here inadequate and we can conclude that MCs themselves need to be proved. It also the conclusion of [323,324] where the authors have done a mechanical verification (using PVS) of automatic theorem prover methods: they note that the inefficiency of fully expansive proof generation can be avoided through verifying the decision procedures[2]. In [325] the author argue that trust need not be achieved at the expense of automation, and outline a lightweight approach where the results of untrusted verifiers are checked by a trusted offline checker. The trusted checker is a verified reference kernel that contains a satisfiability solver to support the robust and efficient checking of untrusted tools. He resumes the problem: *Quis custodiet ipsos custodes ?* (Who will watch the wathmen?[3] that is, who will verify the verifier?). We want to be able to trust the results of provers/model-checkers with a high degree of confidence.

Note that some techniques for abstractions and how finding some protocol specific invariants were describe in [292,301]. But it also limited to constructing proofs of secrecy and authentication properties. In the case of a distributed MC, each machine has only a part of the state-space and gathering all in a single machine is impossible. And if we can consider a distributed COQ, it would be too prone-error for EAL 7. Furthermore, dedicated check tools for evidences can also contain flaws: proving them as well that the evidences do not contain inconsistent uses of axioms is in the same order of magnitude as proving the MC itself.

(c) The Proposed Solution

Another way, proposed in [328] for the PAT MC [[81]] is to use coding assumptions directly in the source code: they indeed use Spec# [230] and a check of the object invariants (the contracts) is generated. They cannot completely verify PAT and they thus focus on safety properties (as no overflows, no deadlocks) of the underlying data structure of PAT (which can run on a multi-core architecture) and not in the final result itself. Note that they can also check that options which may conflict with each other are thus not incorrectly used.

Our contribution in this chapter follows this approach but by currently restrict it to MC algorithms and by using the VCG s WHY and BSP-WHY and by extending the verification to the correctness of the

[2]They also find that the verification of decision procedures has been advocated and practised by a number of researchers quite a long time [311].

[3]Which is also the epigraph taken from "the Satires of Juvenal" to the comic "Watchmen" of A. Moore and D. Gibbons.

final result: has the full state-space been well computed without adding unknown states? Since WHY is not addressing programs, we only focus on algorithms and not on real programs. We can focus on which formal properties need to be preserved and not be obstruct by problems specific to a particular programming language. We thus make an abstraction as imperative algorithms. Even, if most of bugs in model-checker will not be due to wrong algorithms but rather due to subtle errors in the implementation of some complex data-structures and bad interactions between these structures and compression aspects, we must first to check the algorithms to get an idea of the amount of work necessary to verify a true model-checker. Our goal is then a mechanically-assisted proof that these annotated algorithms terminate and indeed compute the expected finite state-space. This is an interesting first step before verifying MCs themselves: it allows to test if this approach is doable or not. This is also challenging due to the nature of model-checking (critical system) and to the algorithmic complexity.

In this work, we only use SMT provers. The correctness of our results depends on the correctness of (1) the WHY and BSP-WHY tools (correct generation of goals) and (2) the results of the SMT solvers. The work of [272] and what has been done in the previous Chapter are a first approach for (1) and the work of [323,324,327] are for (2). In a close future, we can thus have the same confidence on the results of [312] and performances since they are realistic imperative codes.

4.2 BSP Algorithms for the State-space of Security Protocols

In this section, we exploit the well-structured nature of security protocols and match it to the BSP model. 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, to load balance the computations and prove these algorithms.

From Publications 12

This section is a mix of the following publications:

- *in [W2] we describe the* BSP *algorithms for the state-space construction of security protocols;*
- *in [C6] we have performed the benchmarks of their implementation using* PYTHON*;*
- *in [C2] we (mechanised) prove their correction using our tool* BSP-WHY*.*

4.2.1 Definition of the Finite State-space and Sequential Algorithms

(a) Definition of the Finite State-space

In the following, the presented algorithms compute only S (the state space as a set) of the Kripke structure. This is made without loss of generality and it is a trivial extension to compute also T and L. The state-space will be noted StSpace and to represent it in the WHY's logic, we used the following:

```
logic s0: state
logic succ: state → state set
logic StSpace: state set

axiom contain_state_space: ∀ss:state set. StSpace ⊆ ss ↔ (s0 ∈ss and (∀ s:state. s ∈ss → s ∈StSpace → succ(s) ⊆ ss))
```

i.e. defines which sets can contain the state-space. Now ss is the state-space (ss=StSpace) if and only if, the two following properties holds: (A) ss ⊆ StSpace and (B) StSpace ⊆ ss; that is equality of sets using extensionality. Note that we do not use an inductive definition for the state-space: only using this first-order axiom makes the SMT provers proves more proof obligations.

(b) Sequential Algorithm

In order to explain our parallel algorithm, we start with a sequential algorithm that correspond to the usual construction of a state space. Figure 4.3 (left) gives this algorithm in WHY-ML using an appropriate syntax for set operations. All computations in are set operations where a set call known contains all the states that have been processed and would finally contain StSpace.

This algorithm, called "Normal", corresponds to the usual sequential construction of a state-space. It involves a set of states todo that is used to hold all the states whose successors have not been constructed yet; each state s from todo is processed in turn (lines 4−5) and added to known (line 6) while its successors

```
let normal () =
 let known = ref ∅ in
 let todo = ref {s0} in
  while todo ≠ ∅ do
   let s = pick todo in
    known←!known ⊕ s;
    todo←!todo ∪ (succ(s) \ !known)
 done;
 !known
```

```
let normal () =
 let known = ref ∅ in
 let todo = ref {s0} in
  while todo ≠ ∅ do
  {
   invariant (1) (known ∪ todo) ⊆ StSpace
         and (2) (known ∩ todo)=∅
         and (3) s0 ∈(known ∪ todo)
         and (4) (∀ e:state. e ∈known → succ(e) ⊆ (known ∪ todo))
   variant |StSpace \ known|
  }
    let s = pick todo in
        known←!known ⊕ s;
        todo←!todo ∪ (succ(s) \ !known)
 done;
 !known
 {StSpace=result}
```

Figure 4.3. Sequential WHY-ML algorithm (left) and its logical annotated version (right).

are added to todo unless they are known already — line 7. Note the "Normal" algorithm can be made strictly depth-first by choosing the most-recently discovered state (*i.e.* todo as a heap), and breadth-first by choosing the least-recently discovered state. This has not been studied here.

(c) Deductive Verification of the Sequential Algorithm

We need to prove three properties regarding this code: it does not fail, it indeed computes the state-space and it terminates. The first property is immediate since the only operation that could fail is pick (where the precondition is "not take any element from an empty set") and this is assured by the **while**'s boolean condition. Let us now focus on the annotations of this algorithm given in Figure 4.3 (right).

Only four invariants (lines $6-9$) are needed: (1) known and todo are subsets of StSpace; at the end, todo will be empty which ensures (A); (2) these sets are disjoint which ensures that only new states are added to known; (3) and (4) known is as StSpace and when todo will be empty, then it ensures (B). For this algorithm, the termination is ensured by the following variant: |StSpace \ known|. It is ensuring at every step using the fact that the algorithm only adds a *new* state s (line 13) since (known ∩ todo)=∅ (line 7).

Not presented here, but we also do the same work for a classical breadth-first algorithm and for the standard recursive algorithm "deep-first search" (DFS). DFS was the harder to prove since it performs a side-effect within its recursive calls.

Now, all the obligations produced by the VCG WHY are automatically discharged by a combination of the SMT provers — CVC3, Z3, SIMPLIFY, ALT-ERGO, YICES and VAMPIRE. For each prover, we give a timeout of ten seconds. In the following table, for each algorithm, we give the number of generated obligations and how many are discharged by the provers:

algo/SMT	Total	ALT-ERGO	SIMPLIFY	Z3	CVC3	YICES	VAMPIRE
Normal	11	2	10	11	7	3	3
Breadth	31	9	31	28	21	10	10
Dfs	49	22	48	47	40	23	26

One could notice that SMT solvers SIMPLIFY and Z3 give the best results. In practise, we mostly used them: SIMPLIFY is the faster and Z3 sometime verified some obligations that had not be discharged by SIMPLIFY. We also have worked with the provers as black-boxes and we have thus no explanation for this fact.

4.2.2 A Naive Parallel Algorithm

One of the main technical issues in the distributed memory state space construction is to partition the state space among the participating machines. Most of approaches to the distributed memory state space construction use a partitioning mechanism that works at the level of states which means that each single state is assigned to a machine. This assignment is made using a function that partitions the state space into subsets of states. Each such a subset is then "owned" by a single machine. We now show how the "Normal" 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 that this algorithm is efficient despite its simplicity.

```
let naive_state_space () =
  let total = ref 1 in
  let known = ref ∅ in
  let todo = ref ∅ in
  let pastsend = ref ∅ in
    while total>0 do
      let tosend = (local_successors known todo pastsend) in
        exchange todo total known pastsend tosend
    done;
    !known


let exchange (...) =
  let rcv = (BSP_EXCHANGE total tosend) in
    todo←rcv \ known;
    let i=ref 0 in
     while (!i<nprocs) do
      pastsend←pastsend ∪ tosend<i>;
      i←!i+1
     done;
```

```
let local_successors (...) =
  let tosend = ref (init_send ∅) in
  while todo ≠ ∅ do
    let s = pick todo in
    known←s ⊕ !known;
    let new_states = ref ((succ s) \ !known \ !pastsend) in
    while new_states ≠ ∅ do
      let new_s = pick new_states in
      let tgt=cpu(new_s) in
        if tgt=bsp_pid
         then todo←!todo ⊕ new_s
         else tosend<tgt>←tosend<tgt> ⊕ new_s
    done
  done;
  !tosend
```

Figure 4.4. Parallel BSP-WHY-ML algorithm for state-space construction.

(a) BSP State-space Construction

The "Normal" algorithm 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:

```
logic cpu: state → int
axiom cpu_range: ∀s:state. 0≤ cpu(s)<nprocs
```

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 the BSP algorithm of Figure 4.4. 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 a partial view on the ongoing computation.

Function local_successors compute the successors of the states in todo where each computed state that is not owned by the local processor is recorded in a set tosend together with its owner number. The set pastsend contains all the states that have been sent during the past super-steps — the past exchanges. This prevents returning a state already sent by the processor: this feature is not necessary for correctness and consumes more memory but it is more efficient mostly when the state-space contains many cycles.

Function exchange is responsible for performing the actual communications: it returns the set of received states that are not yet known locally together with the new value of *total*. The synchronous 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 — it is mainly the MPI's alltoall primitive. Notice that, by postponing communication, this algorithm allows buffered sending and forbids sending several times the same state. More formally:

$$\text{BSP\_EXCHANGE}(tosend) = \begin{cases} total = \sum_{k=0}^{\mathbf{P}-1} \sum_{i=0}^{\mathbf{P}-1} |tosend[\![k]\!][i]| \\ received = \bigcup_{i=0}^{\mathbf{P}-1} tosend[\![i]\!][\mathbf{pid}] \end{cases}$$

In order to terminate the algorithm, we use the additional variable *total* in which we count the total number of sended states. We have thus not used any complicated methods as the ones presented in [20, 329]. 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. 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.

(b) Verification of this Parallel Algorithm

For care of brevity, we only present the verification of the parallel part of this algorithm and not local_successors nor exchange. BSP_EXCHANGE is axiomatized as a collective operation for BSP-WHY. The exchange procedure is only a permutation of the states that is, in global point of view, only states in arrays have moved and there is no lose of states and a state has not magically appear during the communications. Figure 4.5 gives the BSP-WHY-ML annotate parallel algorithm. We also use the following predicates:

- isproc(i) is defined what is a valid processor's id that is 0≤ i<**nprocs**;

- ⋃(p_set) is the union of the sets of the **p**-value p_set that is $\bigcup_{pid=0}^{\mathbf{p}} \text{p\_set}(pid)$;
- GoodPart(<p_set>) is used to indicate that each processor only contains the states it owns that is ∀i:int. isproc(i) → ∀s:state. s ∈p_set<i> → cpu(s)=i;
- comm_send_i(s,j) is set of sended states from processor i to processor j.

As in the sequential cases, we need to prove that the code does not fail, indeed computes the entire state-space and terminates. The first property is immediate since only pick is used as above and again deadlocks (the main loop contains exchange which implies a global synchronisation of all the processors), we can easily maintain that total (which gives the condition of ending of this main loop) have the same value on all the processors during the entire execution of the algorithm. Let us now focus on the correctness properties.

For the correctness of the main (parallel) loop, the invariants (lines $9-18$) of the main parallel loop work as follow: (1) as in "Normal", we need to maintain that known (even distributed) is a subset of StSpace which finally ensures (A) when todo is empty; (2) as usual, the states to treated are not already known; (3) our sets are well distributed (there is no duplicate state that is, each state is only keep in a unique processor); (4) total is a global variable, we thus ensure that it is the same value on each processor; (5) ensures that no state remain in todo (to be treated) when leaving the loop since total is upper to the size of todo, total is an over-approximation of the number of sent states; (6) and (7) usually ensure property (B); (8) past sending states are in the state-space; (9) pastsend only contains states that are not own by the processor and (10) all these states, that were sent, are finally received and store by a processor.

In the post-condition (line 25), we can also ensures that the result is well distributed: the state-space is complete and each processor only contains the states it owns depending of the function "cpu".

For the local computations, the termination is ensure as in the "Normal" algorithm since known can only grow when entering the loop. The main loop is more subtle: total is an over-approximation and thus could be greater to 0 whereas todo empty. This happens when all the received states are already in known. The termination has thus two cases: (a) in general the set known globally (that is in the point of view of all processor) grows and we have thus the cardinal of StSpace minus known which is strictly decreasing; (b) if there is no state in any todo of a processor (case of the last super-step), no new states would be computed and thus total would be equal to 0 in the last stage of the main loop. We thus used a lexicographic order (this relation is well-founded, ensuring termination) on the total size of the **p** known following with total (which is the same value on each processor) when no new states are computed and thus when no state would be send during the next super-step. At least, one processor can no received states during a super-step. We thus need an invariant in the local_successors for maintaining the fact that the set known potentially growth with at least the states of todo. We also maintain that if todo is empty then no state would be send (in local_successors) and received, making total equal to 0 — in exchange.

With some obvious axioms on the predicates, all the produced obligations are automatically discharged by a combination of the SMT solvers. In the following table, for each part of the parallel algorithm, we give the number of obligations and how many are discharged by the provers:

part/SMT	Total	ALT-ERGO	SIMPLIFY	Z3	CVC3	YICES	VAMPIRE
main	106	74	98	101	0	54	78
successor	46	16	42	41	24	14	32
exchange	24	20	22	23	0	16	15

Now the combination of all provers is need since none of them (or at least a couple of them) is able to prove all the obligations. This is certainly due to their different heuristics. We also note that SIMPLIFY and Z3 continue to remain the most efficient.

4.2.3 Dedicated BSP Algorithms for State-space Computing of Security Protocols

We now considering how optimise (but still correct) the "Naive" algorithm for the case of security protocols. We first present their specifics and then incrementally define three algorithms.

(a) Specific Properties

We assume that any state can be represented by a function from a set of *locations* to an arbitrary data domain. For instance, locations may correspond to local variables of agents, buffers, *etc.*

For these algorithms, it is enough to assume that the following properties hold: (P1) LTS function succ can be partitioned into two successor functions $\text{succ}_{\mathcal{R}}$ and $\text{succ}_{\mathcal{L}}$ (where $\mathcal{R}$ and $\mathcal{L}$ are two disjoint locations)that correspond respectively to transitions upon which an agent (except the intruder) receives information (and stores it), and to all the other transitions; (P2) there is an initial state s_0 and there

```
let naive_state_space () =
  let known = ref ∅       in let todo = ref ∅ in
  let pastsend = ref ∅    in let total = ref 1 in
    if cpu(s0) = bsp_pid then
     todo ←s0 ⊕ !todo;
    while total>0 do
     {
      invariant
     (1) ⋃(<known>) ∪ ⋃(<todo>) ⊆ StSpace
and (2) (⋃(<known>) ∩ ⋃(<todo>))=∅
and (3) GoodPar(<known>) and GoodPart(<todo>)
and (4) (∀ i,j:int. isproc(i) → isproc(j) → total<i> = total<j>)
and (5) total<0> ≥ |⋃(<todo>)|
and (6) s0 ∈(⋃(<known>) ∪ ⋃(<todo>))
and (7) (∀ e:state. e ∈⋃(<known>) → succ(e) ⊆ (⋃(<known>) ∪ ⋃(<todo>)))
and (8) (∀ e:state. ∀i:int. isproc(i) → e ∈known<i> → succ(e) ⊆ (known<i> ∪ pastsend<i>))
and (9) ⋃(<pastsend>) ⊆ StSpace
and (10) (∀ i:int. isproc(i) → ∀e:state. e ∈pastsend<i> → cpu(e)≠ i)
and (11) ⋃(<pastsend>) ⊆ (⋃(<known>) ∪ ⋃(<todo>))
      variant pair(total<0>,| S \ ⋃(known) |) for lexico_order
     }
      let tosend=(local_successors known todo pastsend) in
       exchange todo total !known !tosend
    done;
    !known
    {StSpace=⋃(<result>) and GoodPart(<result>)}
```

Figure 4.5. Parallel BSP-WHY-ML annotated algorithm.

exists a function slice from states to natural numbers (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); (P3) there exists a new function cpu_R from states to natural numbers (a hashing) such that for all state s if $s' \in \mathsf{succ}_{\mathcal{L}}(s)$ then $\mathsf{cpu}_R(s) = \mathsf{cpu}_R(s')$; 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 $\mathsf{cpu}_R(s_1) \neq \mathsf{cpu}_R(s_2)$ 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 $s'|_{\mathcal{R}} = s|_{\mathcal{R}}$ then $s' \in \mathsf{succ}_L(s)$, else $s' \in \mathsf{succ}_R(s)$; where $s|_{\mathcal{R}}$ denotes the state s whose domain is restricted to the locations in $\mathcal{R}$. Intuitively, succ_R corresponds to transitions upon which an agent receives information and stores it.

For example, we can give (part) of these properties in the WHY's logic using the following:

```
logic slice: state → int
(* Two successors functions *)
logic succ_l: state → state set
logic succ_r: state → state set

(* P1 *)
axiom def_cut_succ_1: ∀s:state. succ_l(s) ∩ succ_r(s) = ∅)
axiom def_cut_succ_2: ∀s:state. succ(s) = succ_l(s) ∪ succ_r(s)

(* P2 *)
axiom slice_s0: slice(s0)=0
axiom succ_l_slice: ∀s:state. ∀s':state. s' ∈succ_l(s) → slice(s')=slice(s)
axiom succ_r_slice: ∀s:state. ∀s':state. s' ∈succ_r(s) → slice(s')=slice(s)+1
axiom succ_r_slice2: ∀s,s':state. slice(s')>slice(s) → ¬(s ∈succ(s'))
```

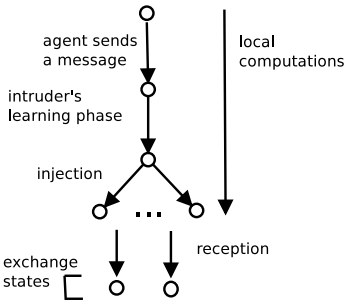
On concrete models, it is generally easy to distinguish syntactically the transitions that correspond to a message reception in the protocol with information storage. Thus, is it easy to partition succ as above and, for most protocol models, it is also easy to check that the above properties are satisfied. This is the case in particular for us using the ABCD formalism.

Note that our approach is independent of using POR reductions as in [288] which is an extension of [303] (to handle security protocols in which participants may have choice points), in both, the main idea is that the knowledge of the intruder always growth and it is safe to prioritise sending transitions — again receptions and local computations of agents. For both, a simple modification of the successors functions is sufficient. This is what we did for the benchmarks.

Based on the following properties, we have designed in an incremental manner, three different BSP algorithms for effectively computing the state space of security protocols. Only the functions local_successors and exchange have been modified in the BSP algorithms.

(b) Increasing Local Computation Time

Using the naive parallel algorithm, function cpu distributes evenly the states over the processors. However, each super-step is likely to compute very few states because only too few computed successors are locally owned. This 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 — scheme in the left. 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 our partition function cpu_R such that, for all states s_1 and s_2, if $s_1|_{\mathcal{R}} = s_2|_{\mathcal{R}}$ then $\mathsf{cpu}_R(s_1) = \mathsf{cpu}_R(s_2)$. For instance, this can be obtained by computing a hash (modulo the number of processors) using only the locations from $\mathcal{R}$.

In this first algorithm, called "Incr", when the function local_successors is called, then all new states from $\mathsf{succ}_{\mathcal{L}}$ are added in todo (states to be proceeded) and states from $\mathsf{succ}_{\mathcal{R}}$ are sent to be treated at the next super-step, enforcing an order of exploration of the state space that matches the progression of the protocol — in *slices*. Another difference is the forgotten variable "pastsend" since no state could be sent twice due to this order. Figure 4.6 (left) schemes this idea.

The invariants are the same of Figure 4.5 but with these changes. First, we need to forget all the behaviour about pastsend in the invariants that is invariants $(9-11)$ since we no longer use this variable. Second, we introduce these two new invariants:

(12) **and** (∀ e:state. e ∈⋃(<known>) → slice(e)<ghost_slice)
(13) **and** (∀ e:state. e ∈⋃(<todo>) → slice(e)=ghost_slice)

We need here to introduce the ghost variable[4] ghost_slice which is incremented at each super-step and thus corresponds to the measure of progression of the protocol. Invariant (12) is needed to prove that the set known contains only states of the past slices and invariant (13) proves that todo contains only states of the current slice. The assertions of the local computations need also to be completely changed. We omit to give them here for care of brevity.

With respect to "Naive", this one splits the local computations, 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 now show how this can be exploited.

(c) Decreasing Local Storage

One can observe that the structure of the computations 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 [330] 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. So we can apply the sweep-line method by making a simple modification of the exploration algorithm. This algorithm call "Sweep" works as "Incr" except that we empty known at the beginning of each super-step. We thus need to maintain this fact by using another invariant (14) ⋃(known)=∅. Note that known only grows in function local_successor and is empty after the communications.

Also, we can thus no longer use known as the variable which contains the full state-space. We thus introduce another ghost variable called ghost_known which will grow at each super-step by recovering all

[4] Additional codes not participating in the computation but accessing the program data and allowing the verification of the original code.

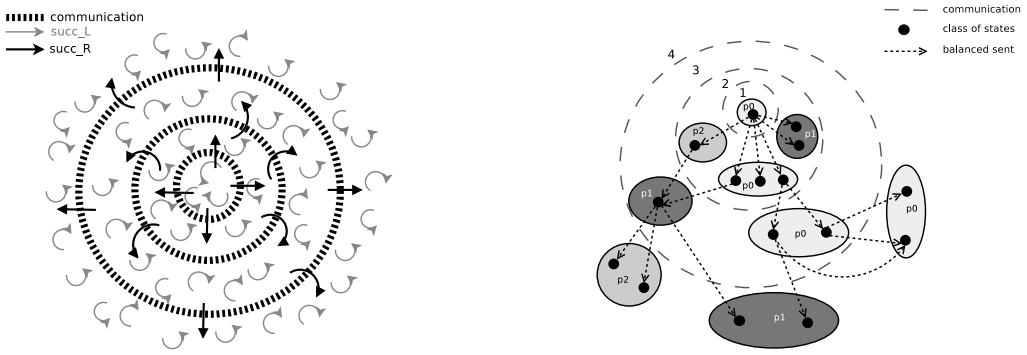


Figure 4.6. Schemes of the "Incr" (left) and "Balance" (right) distributed algorithms.

the states of known. In this way and for having the correctness of this algorithm, in all the previous invariants, ghost_known replaces known.

(d) Balancing the Computations

As the benchmarks performed in [C6] show, those two algorithms can introduce a bad balance of the computations due to a lack of information when hashing only on $\mathcal{R}$. Thus, the final optimisation step aims at balancing the workload. To do so, we exploit the following observation: for all the protocols 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 states 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 $\mathsf{cpu}_R(s) = \mathsf{cpu}_B(s) \bmod P$, where P is the number of processors. This function defines classes of states for which cpu_B returns the same value. Those classes are like a "bag-of-tasks" [331] that can be distributed over the processors independently. To do so, 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 — note that this part of the algorithm has not been proved correct, we only suppose an external parameter that compute how re-distributed the classes. 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. Figure 4.6 (right) schemes this algorithm.

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.

For the correctness of the algorithm, we need a new predicate **class**(e,e') that logically define that two states belong to the same class. We also need to redefine the predicate GoodPart(<p_set>) as follow:

∀i,j:int. isproc(i) → isproc(j) → i≠ j → ∀s,s':state. s ∈p_set<i> → s' ∈p_set<j> → ¬**class**(e,e')"

which denotes that two states that belong to two different processors are not in the same class. We also need to assert that after the computation of the balance (currently axiomatised since it is a heuristic of a NP-problem), sent states respect the predicate GoodPart. We also need the following invariant:

(14) **and** (∀ i:int. isproc(i) → ∀e,e':state. e ∈ghost_known<i> → **class**(e,e') → e' ∈ghost_known<i>)

which denotes that known states respect the fact that all states in a class belong to the same slice at the same processor. The local_successor function should verify this fact in its invariants.

(e) Proof Obligation Results

Notice that for all those algorithms, the termination is proved correct as in "Naive". In the following table, for each part of each parallel algorithm, we give the number of obligations and how many are discharged by the provers:

Algorithm	Part	**Total**	ALT-ERGO	SIMPLIFY	Z3	CVC3	YICES	VAMPIRE
Incr								
	main	109	50	93	85	0	0	85
	successor	105	55	102	101	77	0	73
	exchange	32	15	28	22	19	0	27
Sweep								
	main	129	62	114	109	0	0	92
	successor	107	58	103	102	81	0	78
	exchange	31	14	29	23	21	0	28
Balance								
	main	135	71	123	119	0	0	102
	successor	113	62	111	108	87	0	81
	exchange	38	16	31	29	22	0	29

As above, only the combination of all provers is able to prove all the obligations. And few of them (not necessarily the harder) need that provers run minutes. SIMPLIFY and Z3 still remain the most efficient. An interesting point is that this work has been partially done by a master student: he was able to perform the job (annotated these parallel algorithms) in three months. Based on this fact, it seems conceivable that a more seasoned team in formal methods can tackle more substantial algorithms (of model-checking) in a real programming language.

4.2.4 Experimental Results

In order to evaluate our algorithm, we have implemented a prototype version in PYTHON, using SNAKES 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 [332] for the BSP routines (which are close to a MPI's "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. The benchmarks presented below have been performed using the cluster LACL2. 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) Woo and Lam Pi (WLP) authentification protocol with public keys and trusted server; (5) Kao-Chow (KC) key distribution and authentication.

All are documented at the Security Protocols Open Repository (SPORE) [[82]]. For each protocol, we have built a modular model allowing for defining easily various scenarios involving different numbers of each kind of agents. We note our scenarios NS $-x-y \equiv x$ Alices, y Bobs with one unique sequential session; Y (*resp.* OR, KC, 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:

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. Note that in [C6] we give more details (with graphs) on the differences in behaviour.

By measuring the memory consumption of our various algorithms, we could confirm the benefits of our sweep-line implementation when large state spaces are computed. For instance, in a NS scenario with 5M states, we observed an improvement of the peak memory usage from 97% to 40% (maximum among all the processors). Similarly, for a Y scenario with 1M states, the peak decreases from 97% to 60% (states in Y use more memory that states in NS). 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. We also observed, on very large state spaces, that the naive implementation exhausts all the available memory and some processors start to use the swap, which causes a huge performance drop. This never happened using our sweep-line implementation.

As a last observation about our algorithm, 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 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 (less than 100 states) for which the degree of intrinsic parallelism is very reduced but whose state space is in any way computed very quickly.

4.3 BSP Algorithms for CTL* Checking of Security Protocols

From Publications 13

This section were taken from the following publications:

- *in [C3] we design our* BSP *algorithm for on-the-fly checking* LTL *formula on security protocols;*
- *in [C1] two* BSP *algorithms for on-the-fly checking* CTL* *formula on security protocols;*
- *in [74] which gives the full codes of the algorithms.*

4.3.1 On-the-fly Checking LTL and CTL* Formula

(a) Notations, Definitions, Semantics and CTL* Checking

We now recall the formal definition of CTL*. We will concentrate on the notion of proof-structure [333] 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.

Definition 4 (*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 formula generated from S as state formula and those from $\mathcal{P}$ as path formula. We define the CTL* *formula to be the set of state formula.*

Remark that the CTL consists of those CTL* formula in which every occurrence of a path modality is immediately preceded by a path quantifier and the LTL contains CTL* formula of the form $(A\varphi)$, where the only state sub-formula of φ are literals.
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 5 (*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* $j \leq i$, $x^j \vdash \varphi_2$.

In [333], the authors give an efficient algorithm for model-checking LTL then CTL* formula. 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. It is close to a Tableau method [334]. The rules appear in Figure 4.7. They work on assertions of the form $s \vdash A\Phi$ where $s \in S$ and Φ is a set of path formula.

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 [335] for translating CTL formula into the modal μ-calculus.

Definition 6 (*Proof structure [333]*).
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 7 (*Successful proof structure [333]*).
Let $\langle V, E \rangle$ be a proof structure.

- *$\sigma \in V$ is a leaf if and only if (iff) there is no σ' such that $(\sigma, \sigma') \in E$. σ is successful iff $\sigma \equiv true$.*
- *An infinite path $\pi = \langle \sigma_0, \sigma_1, \cdots \rangle$ in $\langle V, E \rangle$ is successful 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.*

$$\frac{s \vdash A(\Phi, \varphi)}{true} \text{ if } s \vDash \varphi \quad (R1) \qquad \frac{s \vdash A(\Phi, \varphi)}{s \vdash A(\Phi)} \text{ if } s \nvDash \varphi \quad (R2) \qquad \frac{s \vdash A(\Phi, \varphi_1 \vee \varphi_2)}{s \vdash A(\Phi, \varphi_1, \varphi_2)} \quad (R3) \qquad \frac{s \vdash A(\Phi, \varphi_1 \wedge \varphi_2)}{s \vdash A(\Phi, \varphi_1) \quad s \vdash A(\Phi, \varphi_2)} \quad (R4)$$

$$\frac{s \vdash A(\Phi, \varphi_1 U \varphi_2)}{s \vdash A(\Phi, \varphi_1, \varphi_2) \quad s \vdash A(\Phi, \varphi_2, X(\varphi_1 U \varphi_2))} \quad (R5) \qquad \frac{s \vdash A(\Phi, \varphi_1 R \varphi_2)}{s \vdash A(\Phi, \varphi_2) \quad s \vdash A(\Phi, \varphi_1, X(\varphi_1 R \varphi_2))} \quad (R6)$$

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

Figure 4.7. Proof rules for LTL checking [333].

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 sub-goal (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 2 (*Proof-structure and LTL [333]*)

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 successful 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* (SCC) or any accepting cycle. The efficient algorithm of [333] combines the construction of a proof-structure with the process of checking whether the proof-structure is successful using a Tarjan like algorithm for SCC computation (and a recursive decomposition of a CTL* formula into several LTL formula) but a NDFS [336] one could also be used.

Using "proof-structures" is not common and traditionally, LTL checking is perform by the test of emptiness of a Buchi automata which is the product of the LTS and of the formula to check translated to an automata. Generally, a NDFS algorithm checks the presence of an accepting cycle. Our approach "simplifies" the use of our two successors functions and allows us to check CTL* formula without using any (alternative hesitant) automata.

4.3.2 BSP On-the-fly Checking LTL Formula of Security Protocols

(a) On-the-fly Checking Algorithm

As explained in the previous sections, we use two LTS successors functions for constructing the Kripke structure: $\mathsf{succ}_{\mathcal{R}}$ ensures a measure of progression "slice" that intuitively decomposes the Kripke structure into a sequence 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 S_i for all $i < j$. In this way, if we assume a distribution of the Kripke structure across the processors using the cpu_B function, then the only possible accepting cycles or SCC s are locals to each processor. Thus, because proof-structures follow the Kripke structure (rule R7), accepting cycles or SCCs are also only locals. This fact ensures that any sequential algorithm to check cycles or SCC s can be used for the parallel computation[5]. Call this generic algorithm **SeqChkLTL** which takes an assertion $\sigma = s \vdash A\Phi$, a set of assertions to be sent (for the next super-step), and (V, E) the sub-part of the proof-structure that has been previously proceed — this sub-part can grow during this computation. Now, we can design our BSP algorithm which is mainly an iteration over the independent slices, one slice per super-step and, on each processor, working on independent sub-parts of the slice by calling **SeqChkLTL**. This algorithm is given in Figure 4.8.

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 processed at the beginning of each super step; V and E store the proof-structure; super_step stores the

[5]It is mainly admitted that SCC computation gives smaller traces than NDFS [337].

```
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 ParChkLTL((s ⊢ Φ) as σ) is
8    Init_main()
9    while flag =⊥ ∧ total>0
10     send←∅
11     while todo ≠ ∅ ∧ flag=⊥
12       pick σ from todo
13       if σ ∉ V
14         flag ←SeqChkLTL(σ,send,E,V)
15     if flag≠ ⊥
16       send←∅
17     flag , todo,total←Exchange(send,flag)
18   case flag
19    | ⊥ ⟹ print "OK"
20    | σ ⟹ Build_trace(σ)
21
22 def Exchange(tosend,flag) is
23   dump (V,E) at super_step
24   super_step←super_step+1
25   tosend←tosend ∪ {(i,flag) | 0 ≤ i < p}
26   rcv, total ←BSP_EXCHANGE(Balance(tosend))
27   flag , rcv←filter_flag(rcv)
28   return flag , rcv, total
29
30 def subgoals(σ,send) is
31   case σ
32   | s ⊢ A(Φ, p) ⟹ subg←if s ⊨ p then {True}
33       else {s ⊢ A(Φ)} (R1, R2)
34   | s ⊢ A(Φ, φ1 ∨ φ2)
35     ⟹ subg←{s ⊢ A(Φ, φ1, φ2)} (R3)
36   | s ⊢ A(Φ, φ1 ∧ φ2)
37     ⟹ subg←{s ⊢ A(Φ, φ1), s ⊢ A(Φ, φ2)} (R4)
38   | s ⊢ A(Φ, φ1Uφ2) ⟹ subg←{s ⊢ A(Φ, φ1, φ2),
39       s ⊢ A(Φ, φ2, X(φ1Uφ2))} (R5)
40   | s ⊢ A(Φ, φ1Vφ2) ⟹ subg←{s ⊢ A(Φ, φ2),
41       s ⊢ A(Φ, φ1, X(φ1Vφ2))} (R6)
42   | s ⊢ A(Xφ1, ..., Xφn) ⟹
43       subg←{s' ⊢ A(φ1, ...φn) | s' ∈ succ_L(s)}
44       tosend←{s' ⊢ A(φ1, ...φn) | s' ∈ succ_R(s)}
45       E←E ∪ {σ ↦_R σ' | σ' ∈ tosend }
46       if subg=∅ ∧ tosend≠∅
47         subg←{True}
48       send←send ∪ tosend (R7)
49   E←E ∪ {σ ↦_L σ' | σ' ∈ subg }
50   return subg
```

Figure 4.8. A BSP algorithm for LTL checking.

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=⊥).

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 (property P4). SeqChkLTL uses subgoals to traverse the proof-structure. For rules (R1) to (R6), the result remains local because SCC can only be locals. 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 processed in the current step, while the latter results in states to be processed in the next step. If no local state is found but there exists remote states, we set subg←{True} which indicates that the local exploration succeeded (P2) and allows to proceed to the next super step in the main loop. When all the local states have been processed, 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 non-⊥ flag with the lowest dfn value computed on the processor with the lowest number, which allows to ensure that every processor chooses the same flag and then computes the same trace. If no such flag is selectable, filter_flag returns ⊥. To balance the computation, we use the number of states as well as the size of the formula — 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 σ that invalidates the formula is found, a trace from the initial state to σ is constructed. The data to do so is distributed among processors into local files, one per super step. We thus use exactly as many steps to rebuild the trace as we have used to reach σ — not presented in here but more details can be found in [74].

(b) Experiments

We have implemented a prototype of this algorithm using PYTHON and SNAKES again. While suboptimal comparing to a traditional MC, this prototype nevertheless allows an accurate *comparison* for speedup.

In order to evaluate our algorithm, we have used two formula of the form φ **U** deadlock, where 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" (φ is auth**V**learn where auth is an atomic proposition of authentification of the agents and learn the fact that the intruder has broken the secrecy) and Fml2 is the "availability" formula (presented above) [76, 338] – which are common formula for verifying security protocols. The chosen formula globally hold so that the whole proof-structure 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

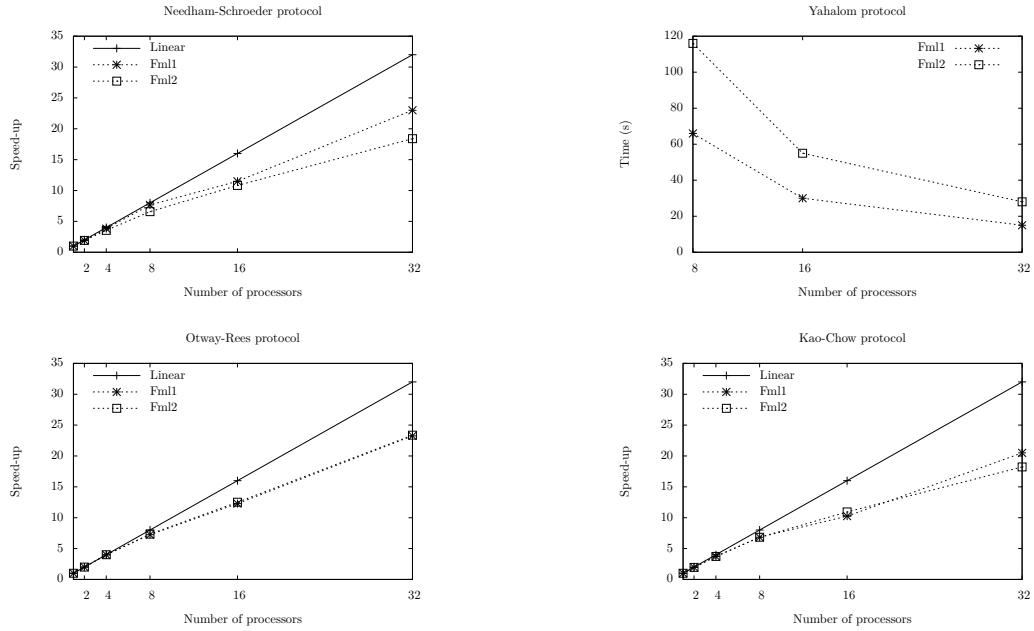


Figure 4.9. Benchmark results for the four protocols where Fml1 is "secrecy" and Fml2 "aliveness".

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 succeeded.

Figure 4.9 gives the speed-up for each of the two formula and two sessions of each protocol. For the Yahalom protocol, the computation fails due to a lack of main memory (swapping) if less that four 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.

4.3.3 BSP On-the-fly Checking CTL* Formula of Security Protocols

For the sake of brevity, we do not present the full codes of the algorithms, they are all available in [74].

(a) A Naive BSP Algorithm for CTL* Checking

The algorithm works as follows. As before, a main loop is used to compute over received assertions and for each of them, a SeqChkCTL* is used to decompose the formulae and run SeqChkLTL adequately to check for an unsuccessful SCC in the proof-structure. We name this computation a "session". During the generation of the proof-structure, when a sub-formulae beginning by **A** or **E** is found (case of rules R1 and R2), the ongoing "session" is halting and is now waiting the result of a new "session" which is running for checking the validity of $s \vDash p$ and the ongoing session is pushed on a stack of waiting sessions.

The main problems are: (1) different processors can throw sessions; (2) a session can induce several super-steps (slices) if it is a path formulae; this is due to the double recursion of the CTL* checking; (3) the different sessions are not fully disjoints; states of the Kripke structures as well as assertions can be shared, that happens when the same sub-parts of the Kripke structure are generated and when sets of formula in the assertions are not disjoints. There is thus no possibility of embarrassingly parallel computations on this set of sessions. A naive solution is to globally select one of these generated sessions (the other still remain in a distributed global stack) and to entirely compute this session until another session is thrown or an answer is find — validity of $s \vDash p$. This naive approach has many drawbacks.

First, each time a session is thrown, it can traverse all the state-space in several super-steps. This can happen when the session has been thrown by a formulae which contains modal operators. This can thus generate too much barriers and induce a poor latency. Second, the sweep-line strategy used in the previous section could not hold: each slice does not correspond to a super-step and thus during backtracking of the answers, the save on disks assertions must be entered in the main memory. This can be also costly. Otherwise, we can keep them all in the main memory but with a risk of swapping. Third,

the balance of the assertions over the processors is done dynamically at each slice of each session: that ensures that two assertions for the same Kripke state would be hold by the same processor. That ensures no duplication of the computations. But if two sessions are run in sequence, the first one will balance some assertions and the second session, if the same states are shared (but for a different set of logical formula) must balance the assertions using this first partial scheme of balance. This is not optimal. A naive solution would be to re-balance all the assertions but it would be too costly.

(b) A "Purely Breadth" BSP Algorithm for CTL* Checking

To avoid these problems we will take into account the "nature" of the proof-structures: having an explicit decomposition of the logical formulae which can help to choose where a parallel computation is needed or not. The main idea of the algorithm is based on the rules R1 and R2: computing $s \vDash \varphi$ together with $s \vdash A(\Phi)$. 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 perform by a LTL session while the computation of $s \vdash A(\Phi)$ would be performed by following the execution of the sequential Tarjan algorithm — for SCC computation. In a sense, we expect the result of $s \vDash \varphi$ by computing the validity of the assertion $s \vdash A(\Phi)$.

We see three main advantages. First, as we computed "simultaneously" both $s \vDash \varphi$ and $s \vdash A(\Phi)$, we would aggregate the super-steps of the both computations and thus reduce to the maximal number of slices of the model. Second, we also aggregated the computations and the communications without unbalance them; similarly, we would have all the assertions of each slice, which implies a better balance of the computations than the use of the partial balances of the naive algorithm. Third, the computation of the validity of $s \vdash A(\Phi)$ can be used later in different LTL sessions. On the other side, the pre-computation of $s \vdash A(\Phi)$ may generate unnecessary works, but, if we suppose a sufficient number of processors, that is not a problem concerning scalability: the exploration is in a breadth fashion that allows us a highest degree of parallelization.

The difficulty in this algorithm is the management of the answers. Indeed, we do not know, *a priori*, the answer of an assertion when we compute the validity of $s \vDash \varphi$ or when it has been send to another processor. Thus, we need to modify the backtracking when an answer is unknown by considering a third possibility of answers: $\perp$ for the case when we cannot conclude. In this way, the "session" is halting until a true boolean answer is computed — mainly in the next slice *i.e.* next super-step.

For the management of the sending assertions, we use two distinct sets. The first one is to store the assertions which are used to continue the exploration of the distributed proof-structure; the second one is for backtracking answers. In this way, at the beginning of a super-step, we first read for answers to possibly unlock halting sessions (store in a stack) which could now continue their works — SCC computations. Then, the algorithm explores the sub-parts of the proof-structures of the received assertions. All these works are done until the initial assertion (of the first session) got its answer. In the case of a flaw, we rebuild the trace as for LTL checking.

In order to keep the sweeping strategy of assertions of the previous algorithms, we must have that states so that assertions do not overlap between different slices. But that does not work cause some assertions do not have their answers (equal to $\perp$) during a slice. So, we can not sweep them into disks when changing of slice. In order to continue to sweep assertions that are no longer needed (they have their answers and belong to a previous slice), we used a variable CACHE which contains all the assertions — LTL sessions is memorised by additional fields in assertions, thus there is no over-cost of memory. At each end of the treatment of a session, we iterate on CACHE to sweep into disk unnecessary assertions, making more main memory available for the next sessions.

(c) Experiments

In order to evaluate our two algorithms in PYTHON/SNAKES, we have test two formula: the first is the LTL formula [76, 338] for testing secrecy of the protocols, whereas the second is CTL and is for fairness [339] (presented above), both formula are common to verify security protocols. The chosen formula globally hold so that the whole proof-structure is computed — and this is widely acknowledged as the hardest case.

In Figure 4.10, we give the speedup of the two latter algorithms ("Naive CTL*" and "Pure Breadth") for three different protocols and for the two formula. As we could expect, the naive algorithm scales less for both formula. Note that for Kao-Chow, both algorithms do not scale well: that is mainly due to a lack of possible attacks which implies less classes of states — the protocol is less parallelized.

Figure 4.11 shows the execution times for our two formula. That has been done for each protocol, with 32 processors. In the figure, the total execution time is split into three parts: the computation time (black) that essentially corresponds to the computation of successful SCC of the proof-structures on

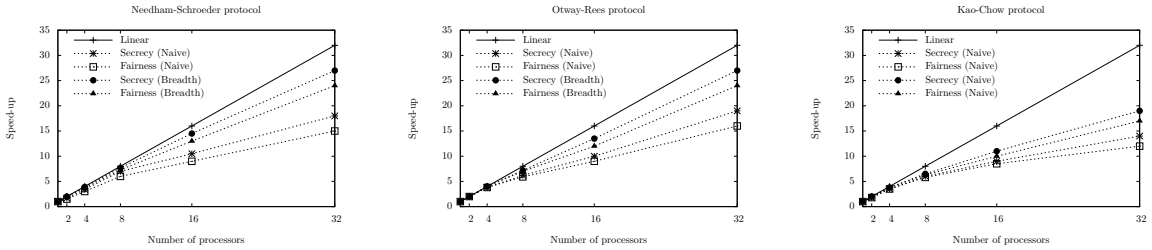


Figure 4.10. Speedup results for three of the protocols.

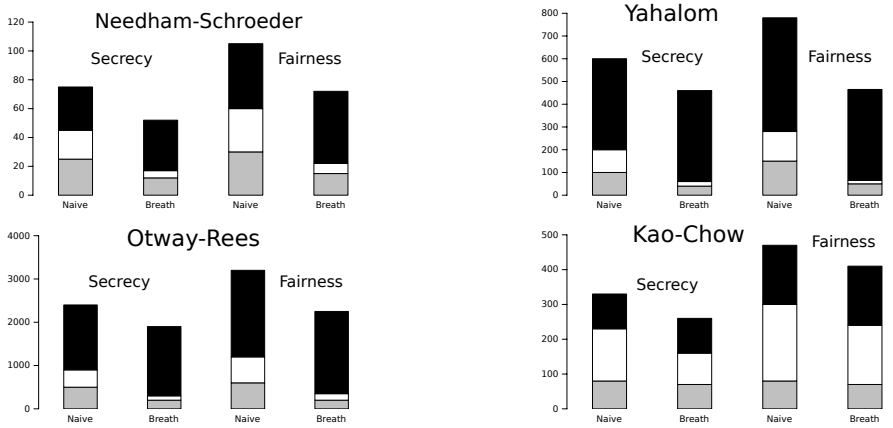


Figure 4.11. Timing of the two algorithms ("Naive CTL*" and "Pure Breadth") and formula (Secrecy and Fairness) for the protocols in seconds where times for the computations are in black, communications in grey and waiting in white.

each processor; the global and thus collective communication time (grey) that corresponds to assertions exchange; the waiting time, *i.e.* latencies (white) that occur when processors are forced to wait the others before entering the communication phase of each super-step. We can see on these graphs that the overall performance of our "Breath" algorithm is always good compared to the naive one. As expected, the "Breath" algorithm reduce both latencies due to less super-steps and a better balance of communications — since they are more *en masse*. Fairness needs more computations since it is a more complicated formulae: the more the formulae and the model are big, the better is the "Breath" algorithm.

4.4 Related Works

4.4.1 Tools and Methods for Security Protocols

Gavin Lowe has discovered the now well-known attack on the Needham-Schroeder public-key protocol using the model-checker FDR [340]. In spite of this, 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, flaws *i.e.* counterexamples), and hybrids that attempt to provide both proofs and counterexamples. In the former, we found dedicated tools as NRL, PROVERIF, SCYTHER, *etc.* [295,296,341], falsification is the domain of model-checking [17,76], and the latter of model-checker with lazy intruders as AVISPA [294, 342]

Most of dedicated tools limit possible kinds of attacks or limit in their model language how agents can be manipulated in ad-hoc protocols. Paper [343] presents different cases study of verifying security protocols with various standard tools. To summarise, there is currently no tool that provides all the expected requirements [16, 343, 344].

It is noticed that less used methods are tests and heuristics for checking flaws in security protocols. But some works exist as [345]. Inspection of messages as a generalist heuristic for reducing the time of verification was studied in [346].

(a) Theorem proving for security protocols

To our knowledge, the first work using theorem proving (HOL) for verifying security protocols is [347]. And now different researches have been conducted in this way, such as [348, 349].

Using a theorem prover, one formalises 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 [350]. Moreover, theorem provers provide poor support for error detection when the protocols are flawed. Even with the work on integrating automatic methods in theorem provers for security protocols as in [298].

(b) Dedicated tools

The first class of tools, which focus on verification, typically rely on encoding protocols as Horn clauses and applying resolution-based theorem proving (without termination guarantee). Analysis tools of this kind include NRL, ATHENA, PROVERIF and MUCAPSL [[83]] — an extension of CAPSL with some language features to handle protocols for secure group management.

For example, the ATHENA [341] tool is an automatic checking algorithm for security protocols. It is based on the Strand Spaces model [293, 351] and, when terminating, either provides a counterexample if the formula under examination is false, or establishes a proof (a PCC) 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 attack within the bound. In the same manner, with the SCYTHER tool, if the security cannot be determined for an unbounded number of agents, the algorithm functions as a classical bounded verification tool and yields results for a bounded number of sessions.

The most known tool is certainly PROVERIF. 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 needed when no flaw has been found in the protocol (with model-checking) and one wants to have a test for an unbounded number of sessions. However it can generates false attacks too. Recently an algorithm was developed in [352] to attempt to reconstruct attacks, in the case when the verification procedure fails.

The three main drawbacks of these tools are (1) the restricted language used for modelling the protocols (in our case, we used PYTHON's expressions as colours of Petri nets, so that we are only limited by PYTHON); (2) the lack of building traces in case of a flaw (this is not the case using a model-checking method); (3) the limit of their verification to simple properties (*e.g.* fairness is not taken into account) and of their model limited to "ping-pong" protocols mainly.

(c) Model-checking of Security Protocols

On the contrary, our approach is based on model-checking that is not tied to any particular application domain. Using CTL*, we can also express many complex properties that some dedicated tool cannot. But that also restrict our approach to finite scenario. Model-checking security protocols is not new [76, 284, 353–358]. This leads to the researches about finding a way to verify whether a system is secure or not. Enumerating model checking is well-adapted too this kind of asynchronous, non-deterministic systems, containing complex data types. In the work of [359], the traditional POR reduction of security protocols is used and in the work of [76], a lazy intruder was used — both are sequential algorithms. Our work has the advantage to take into account distributed architectures.

In [360], the authors have used the MURPHI modelling language [[84]] and different parallel programs for MURPHI now exist such as the one of [361]. But the algorithm [362] uses a naive uniform random hash function — even if the implementation, which uses ERLAN, as in [363], would clearly outperforms our prototype tool due to the use of PYTHON. Notice also the work [364] for the μCRL checker [[85]].

For finite checking scenarios (and enumerate state-space construction), the most well known tool is certainly AVISPA [294]. In contrast, our approach is based on a modelling framework (algebras of Petri nets) with explicit state-space construction, that is not tight to any particular application domain. Our approach, however, relies on the particular structure of security protocols. We believe that our observations and the subsequent optimisations are general enough to be adapted to the tools dedicated to protocol verification: we worked in a very general setting of LTS, defined by an initial state and a successor function. Our only requirements are four simple conditions (P1 — P4) which can be easily fulfilled

within most concrete modelling formalisms.

As an example of application, a distributed memory algorithm with verification of security protocols is described in [365]. The authors use buffering principle and employ a cache of recently sent states in their implementation to decrease the number of messages sent. Unfortunately, the verification of temporal properties is not supported due to the difficulties of combining the parallel checking with the symmetry reduction. [19] allows to verify some properties about some classes of protocols for an infinite number of sessions and with some possibility of replay using a process algebra. Nevertheless, no logic can be used here and each time a new property is needed, a new theorem has to be proved. That could lead to a complex maintenance of the method. Furthermore, the method cannot be applied to some protocols, *e.g.*the Yahalom protocol. On the contrary, our approach is based on a modelling framework with explicit state-space construction, that is not tied to any particular application domain. Using PYTHON in our implementation allows us to manipulate any kind of data-structures that could be used by agents in protocols. Using PYTHON has been shown to be a "good" trade-off between quick modelling and performance [309] and a model compilation approach can be successfully applied to compete with state-of-the-art tools as shown in [366]. So, instead of using a dedicated framework, our approach mainly relies on the particular structure of security protocols. This is a well-desired feature for checking P2P security protocols as in [367, 368] of the SPREADS project — a P2P extension of [369].

4.4.2 Distributed and Parallel Model-checking

To our knowledge, the first work on parallel model-checking is the one of [370]: it used a distributed $n \times n$ matrix representing the graph of the computational model for representing the successor function. The algorithm works by repeating six uses of collective operations (map, reduce, scatter, bcast) and the number of iteration is n. Now, the main idea of most known approaches to the distributed memory state-space generation is similar to the "naive" algorithm of [20] — which can introduce too many cross-transitions. More references can be found in [329] and in [371, 372] for high-level Petri nets. In the latter, various techniques from the literature are extended in order to avoid sending a state away from the current processor if its 2nd-generation successors are local. This is complemented with a mechanism that prevents from re-sending already sent states. The idea is to compute the missing states when they become necessary for model-checking, which can be faster than sending it. That clearly improves communications but our method achieves similar goals, in a much simpler way, without ignoring any state.

Using the summaries of [304, 373], we can find: [374] where different heuristics are used to localise the computations (and thus reducing cross-transitions); [375] where a balancing strategy is used (a balance is performed each time the system detects too many states on a node, which can implies too much communications in our case); [376], which gives details of the design of the EDDY model-checker [[86]] which uses MPI and an explicit breadth search algorithm (but a static hashing partition function) for checking properties written in MURPHI.

To our knowledge, the first BSP algorithm for state-space construction were presented in [377]. The authors limit their study to deadlock and livelock detection of CSP expressions. For termination, an additional global "reduction operation" is performed to discover the total number of pending states which remain in the system. They also used fixed size buckets and processors computed until their own buckets were full, which induce a greater number of super-steps but a better balancing.

Close to our hashing technique, [378] presents a hashing function that ensures that most of the successors are local: the partition function is computed by a round-robin on the successor states. This improves the locality of the computations but can duplicate states. Moreover, it only works well when network communication is substantially slower than computation, which is not the case on modern architectures for explicit model-checking.

For load balancing, [379] presents 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 the 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. We can also cite [380] where remapping is initiated by the master node when the memory utilisation of one node differs more than a given percentage from the average utilisation of all the others.

In [381], a distributed state space exploration algorithm derived from the SPIN model-checker [[87]] is implemented using a master/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, similarly to our function $\mathsf{cpu}_{\mathcal{R}}$. The algorithm performs well on homogeneous networks of machines, but does not outperform the standard implementation, except for problems that do not fit into

the main memory of a single machine. Moreover, no clue is provided about how to choose correctly the fraction of states that has to be considered for hashing, while we have relied on reception locations from $\mathcal{R}$. SPIN has also been used for verifying security protocols [382].

There also exist approaches such as [383], 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.

In [384], an user defined abstract interpretation is used to reduce the size of the state-space, and so it allows to distribute the abstract graph; the concrete graph is then computed in parallel for each part of the distributed abstract graph. In contrast, our distribution method is fully automated and does not require input from the user.

In [385] authors used complex *distributed file systems* or shared *databases* to optimise the sending of the states, especially when complex data-structure are used internally in the states — as ours. That can improve our implementation but not the idea of the method. In [386], the authors used heuristics for the sweep-line method with the drawback that these heuristics can fail. In our case of security protocols, no such heuristic is necessary since the structured model gives the progression. The use of saturation for parallel generation is defined in [387] but improve only memory use and does not achieve a clear speedup with respect to a sequential implementation. In [388], the authors give a parallel algorithm that reduce the state-space using bi-simulations. However, it is not clear how this technique can be used in our case.

Very close to our idea of localising cycles, we can cite [389] and [390] which both used partition functions that enable cycles to be local only — as for us. The limits of the method are the cost of their functions as well as the number of SCC s which is not sufficient to scale. [391] presents distributed algorithms for SCC computation as the OWCTY "The One Way Catch Them Young algorithm" [392], NEGC "Negative Cycle Algorithm" [393], MAP "Maximal Accepting Predecessor" [394], *etc.* In our work, all SCC s are purely local, which is easier to handle and more efficient.

For multi-cores and GPUs (shared memory architectures), different works exist. In [395], the author argues for designing algorithms for hybrid machines (cluster of multi-cores). Our algorithm fits naturally this kind of machines, since classes of states (potentially several on each node) are independent. In [396,397], the authors propose multi-core algorithms for NDFS. For CUDA (GPU), a specific implementation of NDFS checking has been done in [398]: to fit these specific architectures, a matrix representing the successor function has been used.

A kind of tree (hesitant) Büchi automata is used in [399] where parallel SCC computations are performed. The automata is hesitant is the sense that as for rules R1 and R2, it cannot conclude and thus initiates the two possible computations. That generated what they call "games" (close to our "sessions") and the algorithm has to manage how to store partial results of games. Shared memory computations and heuristics are used here to simplify this management. The algorithm has also expensive management of invalid SCC s which seems not feasible for a distributed architecture. Those algorithms was also test to check security protocols in [339]. The same kind of automata were used in [400] for checking CTL* formula on a network protocol called "Rainbow". If parallel model-checking of LTL formula has been the more studied, works for CTL [401] and the μ-calculus (which is more expressive than CTL*) can be found in [402–404].

4.4.3 Verifying Model-checkers

To our knowledge, there are four existing approaches to automatically generate machine-checked protocol security proofs. The first approach is in [299] where a protocol and its properties are modelled as a set of Horn-clauses and where the certificate is machine-checked in COQ. The second [298] uses the theorem prover ISABELLE and compute a fixpoint of an abstraction of the transition relations of the protocol of interest — this fixpoint over-approximates the set of reachable states of the protocol. The latter [291,292] also uses ISABELLE and invariants are derived from an operational semantics of the protocols. We see three main drawbacks to these approaches. First, they limit (reasonably) protocols and properties that can be checked. Second, each time, the proof of the tested property of the protocol needs to be machine-checked; in our approach, the results of the MC are correct by construction. Third, there is currently no possibility of distributed computations for larger protocols — hours of computations are needed for some protocols. But they have one evident advantage: only the final result has to be correct (intermediate computations do not need to be verified) and it saves time for correct design (machine-checked) of algorithms.

Some developments using theorem provers are related to model checking. In [321] and [322], authors present development of BDD s and tree automata using COQ. The verification of a μ-calculus computation has also been done in COQ in [312]. A sequential state-space algorithm (with a partial order reduction) has been checked in B in [405]. Our methodology is also based on perfect cryptography. The author

of [406] annotated cryptographic algorithms to mechanically prove their correctness.

A mechanically assisted proof, using ISABELLE, of how LTL formulae can be transformed into Büchi automata is presented in [407]. Temporal logic in COQ are also available: LTL in [408] and CTL* in [409]. All these works are of interest since logical theories may be axiomatised in WHY.

In the same way, model compilation is one of the numerous techniques to speedup explicit model-checking: it relies on generating source code (then compiled into machine code) to produce a high-performance implementation of the state-space exploration primitives, mainly the successor function. In [410], authors propose a way to prove the correctness of such an approach. More precisely, they focus on generated Low-Level Virtual Machine (LLVM) code from high-level Petri nets and manually prove that the object computed by its execution is a representation of the compiled model. If such a work can be re-made using a theorem prover, we will have a machine-checked successor function which is currently axiomatized in WHY. Furthermore, the Dolev-Yao model assumes a perfect cryptography; a mechanised proof of some cryptographic primitives has been done in [406].

4.5 Lessons Learnt from this Work

Design and implement a parallel model-checker is a huge work, even if it is only a prototype. Security protocols are hard to analyse and contain many subtleties. After having co-directed a thesis in this subject, I realise that many things have to be studied. I still learn a lot and still have much to learn. This experience allowed me to "understand" security protocols, LTL/CTL* model-checking, model using Petri-Nets, *etc.*

First lesson, even if my colleague Pr Franck Pommereau would not agree (he is the author of SNAKES and ABCD), I think that for parallel model-checking "PYTHON sucks"[6]. It is too unsafe (just dynamically typed), undeterminist (*e.g.* hashing objects and list traversal which can differ on nodes and thus introduce bugs due to underterministic results) and very slow. The only advantage for algorithm design is its friendly syntax and its very stable performances: run many times, a program would have always the same execution time. To implement another model-checking algorithms, I will never re-use PYTHON. Too unsafe. For now, this slow has prevented us from comparing our work with existing tools even if the results of [366] are encouraging. The reader could point off that why not having used BSML? A binding between SNAKES and OCAML can be used. The reason is that these bindings would be too slow, unsafe and the current bindings do not work well.

Second, we have designed our algorithm in such a way that they can be "easily" proved and in a step-wise way. This has been (mechanised) the case. It is nice to see the built of a model-checker that is (mechanised) correct from the beginning. I believe that this kind of work (for many kinds of software) will (should?) be the norm in a close future, even if it requires more time and money. And there are now too many circumstances where the fidelity of the verification claims does matter.

[6] In [411], the authors argue that even if functional programming is suitable for implementing model-checking tools, imperative features and a strict semantics (not the lazy one of HASKELL) for performance are needed; ML seem a good choice. The authors also raised the question of the relevance of the use of PYTHON, because of its apparent simplicity. My current experience makes me answer no.

5 Conclusion

I hope that the modest contributions presented in this document will convince the reader that the BSP model is viable for a large number of areas ranging from functional programming to model-checking.

5.1 Summary of the Contributions

Parallel programs have applications in many different areas. They offer the possibility of computing data much faster than sequential programs, which is important in applications. However, even small parallel programs are difficult to design, and errors are more often the rule, not the exception. *Verification* is the rigorous demonstration of the correctness of a program with respect to a specification of its intended behaviour. *Formal methods* are increasingly being applied to software and programs. These applications create a strong need for on-machine formalisation and verification of programs — programming language semantics, protocols, compiler, OS, *etc.*

Since the paper "Goto Statement Considered Harmful", *structured* sequential programming is the norm. And verification of sequential programs is an old subject where some results are now relevant, as in [117]. But it is surprising to see that it is absolutely not the case for parallel programming. We reminder that we see three main reasons for this fact.

First, besides compiler-driven automatic parallelisation, programmers have kept the habit to use low-level parallel routines (as asynchronous send/receive of *e.g.* MPI) [6] or concurrent languages [7]. In this way, they more or less manage the communications with the usual problems of (un)buffered or (un)blocking sending, which are sources of *deadlocks* and *non-determinism*. That makes the verification of parallel programs harder since the verification tool has to manage the common concurrent drawbacks — data race, deadlock, interleaving of the instructions, *etc.* Programmers (see them as "engineer prover") cannot focus on the essential: correctness of the algorithms in addition to the detection of typical bugs such as buffer/integer overflow. Furthermore, they forbid optimisations that could be done if more *high-level structures* such as collective operators or skeletons were used instead.

Second, there is not "standard" of parallel computing and two many paradigms exist — except for the standard MPI. Any researcher has first to enter the world (folk) of parallel computing before he is able to produce any result. Perhaps because we do really not known what parallel computing is. We can see that this problem is much less prominent in sequential programming languages: exagerating a little, any new result for C will have its equivalent to JAVA, much faster than in the parallel computing world.

Last, high-level models, languages and verification tools for parallel computing are needed but, unfortunately, rarely used. The main reason for that is they do not generally offer a sufficiently wide set of parallel structures (as skeletons [77]) for a practical and efficient programming.

This makes the design of new and robust parallel languages and tools an important area of research.

5.1.1 Functional BSP Programming

There is much more research into advanced programming paradigms required to find the best ways to accommodate parallelism. In my own doctoral thesis, followed by the one of Louis Gesbert (that I co-directed under the PROPAC project with Pr. Frédéric Loulergue), we have studied BSML, which is a high-level BSP language. BSML has an explicit parallelism and a cost-model. It allows safe execution of programs and it is possible to generate certified programs using the COQ proof assistant. It therefore, overcomes the problems described above, except for two points: (1) it is still another programming language for the *Tower of Babel* of the programming languages; (2) it "only" allows program BSP algorithms.

In this work, we first designed a minimal set of primitives for BSP computing. This set of primitives was axiomatized in the COQ theorem prover: that allowed us to extract machine-checked parallel functional programs from specifications — in [R8] some simple examples have been proved and [73] contains a bigger example: a data-parallel tree skeletons.

We then extended the language with the *common features of* ML programming: exceptions, imperative features, I/O operations and pattern-matching. A parallel composition primitive has also been added. For each extension, an (operational) *semantics* study has been performed to formally find what kind of instructions can break the essential property of BSML which is confluence. Furthermore, an implementation has been done to perform *benchmarks* of some non trivial applications. Different semantics have been used. And they have been proved equivalent to the execution of a *parallel abstract machine* which contains only a few set of parallel instructions. An implementation (using a low level library) of this set is only needed for the runtime of BSML. And thus this implementation has only needs to be (mechanised) proved correct for a correct implementation of BSML.

Finally, a *type system* has been designed to prevent run-time problems due to the incorrect use of the parallelism. And the type system is inherently compatible with all the presented features.

This makes BSML a concrete and serious candidate for parallel computing.

5.1.2 Deductive Verification of BSP Algorithms

As it is difficult to prove everything in COQ and, for efficiency reasons, some algorithms (such as model-checking) cannot be programmed by purely functional programs, it thus seems desirable to go through the *deductive verification* of parallel iterative programs. In the doctoral thesis of Jean Fortin (which subject comes from the PROPAC project), we present a methodology and its associated tool, called BSP-WHY for deductive verification of BSP algorithms. BSP-WHY is an extension of the WHY's intermediate language. BSP-WHY adds some constructs specific to BSP parallelism and it *translates* a subset of the BSP-WHY codes (those that are "well-structured" enough) into plain sequential ones, that can be used as an input for various tools such as VCG.

Since BSP uses barrier synchronisation, parallelism can be removed by replacing a portion of code between barriers, with a loop to repeat that portion for every process. We think that building upon an existing tool for program verification (and not doing this work from scratch) is quite appealing since generated proof obligations require a substantial work — both in theory and practise. Some examples are given and also the number of automatically discharged proof obligations. In view of this first ratio "number to prove/is proved automatically" we believe this method is far from perfect — notably for correctness. Nevertheless, confidence in the code can be quickly increasing since reliability and safety issues are automatically verified.

Lastly, BSP-WHY were used to prove *distribute state-space algorithms* as a first step towards "checking the verifier". Model checkers are specialised software, using sophisticated algorithms, whose correctness is vital. In this work, we focus on the correctness of well-known sequential algorithms for finite state-space construction (as the basis for explicit model-checking) and on the distributed ones designed by the authors. We annotated the algorithms for operations on finite sets, in order to set goals that were entirely checked by SMT solvers. These goals ensure the termination of the algorithms as well as their correctness for any "successor" function — assumed correct and generating a finite state-space. We thus gained more confidence in the code. We also hope to have convinced that this approach is feasible and applicable to real (parallel or not) model-checking algorithms.

Given the limitations of BSP-WHY, we certainly cannot claim that our method can be used to verify the correctness of every BSP program. But we have shown that it works on some interesting, non-trivial examples and that when it is applicable, it seems to be an effective approach to deal with a difficult problem.

In addition, we have designed *mechanised operational semantics* for iterative BSP programs[1] An "originality" of this work is that the semantics have been written in COQ. If it is the norm for semantics of sequential languages and it is increasingly used in concurrent languages, for parallel computing (HPC), it is still at its first faltering steps.

The main goal of this work is to propose a framework where programmers can prove correctness of their BSP programs and, finally, automatically get high-performance versions in a certified manner.

5.1.3 BSP Algorithms for the Verification of Security Protocols

This work was done during the doctoral thesis of Michael Guedj (co-directed with Pr. Franck Pommereau) under the SPREADS project. It has allowed me to put into practise the work of deductive verification. The explicit model-checking of security protocols is also a good example of HPC because it is a symbolic

[1] In [C8], we tried to introduce high-performance primitives: we have given a simple transformation of the source code that generated some calls to high-performance routines in place of BSP classical ones; *i.e.* transforming some buffered operations into unbuffered ones as in [412]. An executable can be obtained by automatic extraction of executable OCAML code from COQ.

and irregular application which consumes huge ressources. In addition, security protocols are used in HPC because parallel machines are often remoted and shared: therefore, it seemed necessary to be able to communicate with them in a secure way.

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. There are now many tools that check the *security* of cryptographic protocols and *model-checking* is one solution [10]. It is mainly used to find flaws in *finite scenario* (bound number of agents) but not to prove the correctness of a protocol. But explicit model-checkers use sophisticated algorithms, that can miss a state which can be an unknown attack on the security protocol. *Mechanised correctness* is thus important since missing a flaw or finding a false flaw can waste a lot of time for engineers — at least.

We have thus designed different parallel algorithms in such a way that the mechanical correctness be feasible — currently, only the state-space algorithms are proved correct. My work contributes to the development of such techniques, by modeling security protocols using an *algebra of coloured Petri* net called ABCD and reduces the time to check the protocols, using parallel computations. This allows parallel machines to apply automated reasoning techniques, to perform a formal analysis of security protocols. Moreover some of the parallel algorithms have been mechanised prove correct for increasing confidence in their results.

To check if protocol scenario contains *flaw* or not, we propose to resort to *explicit distributed model-checking*, using an algebra of coloured Petri nets to model the protocol, together with security properties that could be expressed as reachability properties, LTL, or CTL* formula. Reachability properties lead us 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) which is combined with the formula under analysis into a so-called proof-structure. In both cases, *on-the-fly* analysis allows to stop states explorations as soon as a conclusion is drawn.

Our parallel algorithms use the *well-structured* nature of the protocols in order to choose which part of the state-space is really needed for the partition function and to empty the data-structures in each super-step of the parallel computation. They also entail automated classification of states into classes, and the dynamic mapping of classes to processors. We find that both of our methods execute significantly *faster* than others and achieve a better network use. Our method can also work for other problems than protocols.

In the case of LTL, we have seen that no cross-transition occurs within a strongly connected component, which is crucial to conclude about formula truth. In the case of CTL* however, local conclusions may need to be delayed until a further recursive exploration is completed, which might occur on another processor. Rather than continuing such an exploration on the same processor, which would limit parallelism, we designed a way to organise the computations so that inconclusive nodes in the proof-structure can be kept available until a conclusion is drawn 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.

5.2 Future Works and Perspectives

5.2.1 Ongoing Works

We outline here some of the immediate perspectives of the works presented in the above chapters; each of them may be the topic of a doctoral thesis.

(a) Parallel Functional Programming

The flat view of a parallel machine as a set of communicating sequential machines (as BSP reminds us) remains useful, but it is nowadays incomplete [3]. For example, GPU processors have a master-worker architecture; clusters of multi-cores make the cores share a few network cards, which implies a bottling and a congestion of the communications. We can also consider clusters of clusters (also known as *departmental meta-computing*), which was the subject of [R5] where we proposed to have a two levels approach: a BSP one on clusters and a more asynchronous one for the management of the clusters. Two kinds of parallel vectors were used: one for the BSP computations (as BSML) and one for the management of the clusters.

With these issues in mind, the authors of [413] proposed a language called SGL (for scatter/gather language) which is a master-worker language based on the multi-BSP model [414]: a multi-level variant of the BSP model — to our knowledge, the first multi-BSP model were proposed in [415]. The one of [414] is

more realistic for "cluster of clusters of multi-cores with GPU processors". This recursive model abstracts parallel architectures as trees of processes (each branch is either a master and a child, except for leaves) and thus needs recursive algorithms. In [C4], we have used SGL to implement and benchmark (on a cluster of multi-cores) several data-parallel skeletons. But SGL does not seem appropriate for more complicated algorithms such as the ones for model-checking. By allowing only scatter and gather operations, it is not possible to have point-to-point communications.

Another finding concerns the primitive **put** used to describe communications in BSML: it is too hard to use (students who have tried BSML have often been blocked by it) and it does not easily allow the sending of messages while continuing the calculations — messages that will be accessible to the next super-step. This is the case of our model-checking algorithms — even though we have not used this option because it is not available in BSP-PYTHON. For these reasons, I propose to completely change BSML— although this is nethier yet implemented nor studied semantically, and is currently just an idea.

First, we should have send/receive primitives that are buffered and can only be local — within a parallel vector. This breaks the BSML "philosophy" which was to avoid send/receive primitives. These primitives are lower level and not functional at all; but they can easily simulate the classic **put**, which is well suited to COQ — for certified BSML programming.

Second, to avoid the problem of messages that are in the communications environment (which make BSPLIB primitives non-compositional as explained in Chapter 3), I propose the explicit and dynamic creation of groups of processors and of communications environments. At the beginning of the BSP computation, as the MPI_COMM_WORLD, an initial empty environment exists which contains all processors. The BSP synchronisation would only be performed by the **proj** primitive and would occur for a list of environments. A generic broadcasting would look like:

```
let bcast v from set_procs =
 let new_env = set_procs.newComm in
  ≪ if $pid$ = from
      then
       for i=0 to set_procs.nprocs() do
        send i $v$ new_env
       done ≫ _{set_procs};
  proj [new_env] ≪ ()≫ _{set_procs};
  ≪ received new_env from ≫ _{set_procs}
```

Notice that the second parameter of **proj** can be ignored and be an optional argument for extracting values of a parallel vector. The variable set_procs is the current group of processors — here designed as an object. Vectors are also indexed with a variable: a group of processors. In case of a pure BSP computation, this index can be omitted to find our classic BSML. This index would be used to avoid using a vector with another one which "work" for a different group of processors. Testing if a vector (*e.g.* v in the above example) is indexed for the correct group of processors seems to be a variable scope analysis. The creation of a new group of processors would be done using a specific primitive. Note that all primitives that manipulate environments or groups are prohibited inside vectors — this represents an easy check for our type system.

For multi-level BSP computing, two solutions are to be tested. First, an explicit constructor inside vectors that allows nested parallel vectors: ≪ NESTED(≪ e≫) ≫ . Communications from a node to its children would be perform during this nested construction. As in SGL, a specific operator (**master**) could return the boolean condition to known if a node had children or not — in the latter case, the creation of a parallel vector would be executed as a single expression. This is the solution chosen in [134] for BSP ++. Second, as in SGL, a **pardo e done** constructor would run e on each children.

Currently both solutions are unsatisfactory because the interaction with nested vectors (or pardo) and with environments and groups of processors is not clear. It leads to some questions: can we avoid the use of a group of processors in the nested vectors when recursive algorithms are used? Can we avoid incorrect used of parallel vectors that were created in different levels of the tree?

(b) Deductive Verification of Parallel Programs

First, we intend to add a companion tool for C or JAVA programs, as in [220]. The tool for C programming is a plug-in called JESSIE for the FRAMA-C framework [[88]] — note that another plug-in call WP is also available but it directly generates conditions without using WHY. JESSIE generates WHY codes from C ones. For JAVA programs, the KRAKATOA tool will be considered — it also generated WHY codes. For ADA programs, we are waiting for a first public tool.

We need to test our method on realistic BSP computations, even if the results of our examples are encouraging. Programs of [25, 31] will be used. JAVA programs (that use HAMMA [130]) will be perhaps considered that HAMMA's communication primitives are very close to those of BSP-WHY.

Second, BSP is an interesting model because it features a cost model for an estimation of the execution time of its programs. Giving these costs formally by extended pre-/post-condition and invariants is an interesting challenge — one could speak of a cost certification. In fact, many scientific algorithms (numeric computations such as matrix ones) are not too complex; it is often a polynomial number of super-steps. This is an ongoing work of the doctoral thesis of Jean Fortin.

Third, conditions generated by WHY from BSP-WHY are not friendly. In automatic theorem provers, one does not see them at all but, in case of manual proofs, that could be problematic. This is mainly due to the massive use of the generated **p**-loops: special tactics are needed to simplify the analysis. In the same manner, syntactic sugar is needed to manipulate the communications environments (list of messages) as an "user library" to facilitate the writing of logical assertions.

Fourth, there are many more MPI programs than BSP ones. Our tool is not intended to manage all MPI programs. It cannot be used to model asynchronous send/receive ones — with possible deadlocks depending on the MPI scheduler [210, 211]. Only programs which are BSP-like (*e.g.* MPI's collective primitives) [9] will be considered. Analysing MPI programs to find which are BSP-like and to interpret them in BSP-WHY is a great challenge, as in [125]

Last, changing BSP-WHY in accordance with the wanted BSML definitely worthes considering. Currently, the tool automatically detect which parts of the code are "parallel" or purely sequential. Using vectors will simplify this work. Furthermore, it will work for multi-level BSP programs. In case of JAVA/C programs, it could be the work of the plug-ins to find which parts of the code are parallel or not and to automatically generate the appropriate vectors. We will take this into account for the design of BSML.

(c) Verification of Security Protocols

Future works will be dedicated to build an efficient implementation from our prototypes. Using this implementation, we would like to run benchmarks in order to compare our approach with existing tools as AVISPA. We would also like to test our algorithm on parallel computers with more processors in order to confirm the scalability observed on 40 processors. More practically: we would like to have a tool able to translate HSPSL models [76] (a standard language for describing security protocols) to ABCD ones since HSPSL is mainly used by the community; It will also allow us to test our algorithm on other attackers than the "pure" Dolev/Yao one — considering equational theories as "xor" apply to cryptographic messages. To optimise performances, using a specific library as DIVINE [416] would also have to be considered. This would make benchmarks performing easier.

Another way to improve performances would be to consider symbolic state-space representations, as well as symbolic state space computation [417]. In the former case, we are particularly targeting representations based on decision diagrams. In the latter case, we are thinking about adapting symmetry methods (currently only a partial order reduction is provided),in order to reduce the number of executions that need to be explored. Reduction methods usually result in an exponential reduction of the number of computed states or transitions. On the other hand, using symbolic representations seems more challenging because storing a 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 all our algorithms. Proving a verification algorithm is highly desirable in order to certify the truth of the diagnostics delivered by such an algorithm.

Finally, we would like to generalise our present results by extending its application domain to more complex protocols with branching and looping structures, as well as complex data types manipulations. In particular, we would consider protocols for secure storage distributed through peer-to-peer communication [367] of the SPREADS project, mobile Ad Hoc networks [418, 418–420], secured routing protocols [421, 422], multi-party protocols [423, 424], sensor networks [425] and voting protocols [285–287].

5.2.2 Projects

(a) Parsival Project

This section is the summary of a project we submitted (for funding) to the ANR "Young Researchers" with Joel Falcou (LRI, University of Paris-South) and Ludovic HENRIO (OASIS team, I3S/INRIA).

In the PARSIVAL project, we will focus on the definition of a new hybrid language for parallel programming based on the merging of two famous parallel programming models: BSP and algorithmic skeletons for subset of C ++ — mainly C +templates[2]. Our main goal is to provide a complete framework com-

[2]Templates can be seen as a Turing-complete, purely functional sub-language; C ++ template meta-programming is mainly a monomorphisation of the template code to emulate polymorphism that is duplicates polymorphic functions for

posed of a C ++ library for parallel programming, a logical annotation system of the code for proof with the needed tools for this verification and a machine-checked transformation of these hybrid programs into programs using a low-level library such as MPI.

In the context of hierarchical computing (clusters of multi-cores/GPUs), we claim that the fusion of BSP and skeletons is possible and would solve the abstraction/efficiency trade-off while providing proof of correctness of programs and a safe framework for their execution. Various challenges are still to be solved:

- Neither BSP nor algorithmic skeletons are sufficiant to answer all the needs expressed by modern parallel applications. If BSP is able to handle a large spectrum of data-parallel structures, control structures are usually hard to design and perform poorly. Skeletons, on the other hand, are often forced to fall back to some ad-hoc parallel code, so they can handle complex data-parallel structures. Merging both models is then very sensible.
- Overhead induced by high-level abstractions are often seen as limiting. The target libraries (such as MPI) in which our language will be compiled is then a concrete constraint, as it should deliver high performance and still be able to handle legacy code or third-party systems.

The first step of this project is to design a hybrid language with these two already aforesaid paradigms — in the spirit of [113]. A mechanised semantics will also be needed for correct implementation. Second, we have to build a tool to prove the correctness of programs. The structured nature of the language will be used, not to build it from the scratch, but by using tools for sequential program proofs — thus simulating these structured parallelisms as we already did for BSML in COQ and BSP-WHY. Finally, we need a machine-checked correct compilation scheme to execute them safely: generated adequate MPI code. For the BSP part, we will used traditional MPI implementation of BSP. For the skeletons, we will machine-check the work of [153, 426]. A preliminary study of a MPI operational semantics is certainly needed. We insist on the fact that this work will be realised on a small and adequate subset of MPI, making it feasible as part of our project.

The choice of C ++ is also due to its large diffusion, and its large number of scientific libraries. C ++ and templates (and skeletons) have also been studied for a long time by Joel Falcou while BSP is more my speciality and formal method for skeletons, Ludovic Henrio's domain.

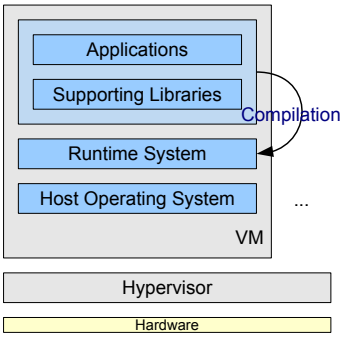
(b) Towards Verified Cloud Computing Environments

From Publications 14

This section (as a call for collaborations) is a summary of [C5].

With the development of mobile and Internet applications, cloud computing becomes more and more important. Increasingly, our data is in the cloud and this use becomes pervasive in our lives. It is therefore necessary to ensure the reliability, safety and security of cloud environments.

A program runs in an environment: when a program is proved correct in its source form, the proof is done with respect to the semantics of the source language. If the compiler is incorrect, even a proved program can go wrong. A verified compiler makes some assumptions itself about the operating system, and so on. The Trusted Computing Base or TCB is the software (and hardware) that is assumed to be correct, without having proved its correctness, in a computing environment. The trust of an environment increases as the size of the TCB decreases.



A usual software stack in cloud computing environments is depicted in left. We argue in [C5] (and give more details) that verification at all these levels is possible and existing work makes us think it is manageable, even if it cannot be reached by a lonely team effort. We discuss each level of the software stack in [C5].

At the application level, we will, first, only consider structured applications such as MAPREDUCE or HAMMA. Then, to execute the applications, we need to compile them: we will consider here only compilation towards byte-code for the JAVA Virtual Machine (JVM) [427] as JAVA is popular for cloud computing platforms in particular MAPREDUCE. Third we need a run-time system: this system consists of a JVM but also of compiled versions of the supporting libraries such as MAPREDUCE. Finally, the JVM runs inside an operating system, itself embedded in a virtual machine running on top of

each used type in the program; Such a simple optimisation is generally sufficient to generate efficient code but needs to be formalised — this simple but crucial task for realistic applications is also a part of this project.

an hypervisor[3]. We will consider here a special case of JVM, such as Oracle's JRockit Virtual Edition [[89]] , that can run directly on top of the hypervisor, without a host operating system. In our stack of verified software, it means that the only remaining piece of software to deal with is the hypervisor.

In an usual software stack of a cloud computing environment, previous work in other context shows that it is possible to formally verify all the layers of the stack. Some of the components to be proved correct are very specific to cloud computing: the applications, the supporting libraries implementations, extensions to the JVM, the hypervisor. Some are not: the JAVA compilers, the JVM implementation.

This work therefore requires the cooperation of several teams that the authors of [C5] relates.

[3]Hypervisors virtualise the underlying architecture, allowing a number of guest machines (partitions) to be run on a single physical host. They represent an interesting and challenging target for software verification because of their critical and complex functionality [428].

Publications

International Journals

[R1] 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*, 26(4):(251–273), 2011. Extended version of [W5].

[R2] L. Gesbert, F. Gava, F. Loulergue and F. Dabrowski, Bulk Synchronous Parallel ML with Exceptions. *Future Generation Computer Systems*, 26(3):(486–490), 2010. Extended version of [C13]

[R3] F. Gava. A Modular Implementation of Parallel Data Structures in BSML. *Parallel Processing Letters*, 18(1):(39–53), 2008.

[R4] F. Gava. External Memory in Bulk-Synchronous Parallel ML. *Scalable Computing: Practice and Experience* (previously named *Parallel and Distributed Computing Practices*), 6(4):(43–70), 2005. Extended version of [C16]

[R5] F. Gava and F. Loulergue. A Functional Language for Departmental Metacomputing. *Parallel Processing Letters*, 15(3):(289–304), 2005.

[R6] F. Gava and F. Loulergue, A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):(665–671), 2005. Extended version of [C22]

[R7] F. Loulergue, F. Gava, M. Arapinis and F. Dabrowski, Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, ACIS, 5(3): (182–199), 2004. Extended version of [C19].

[R8] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):(365–376), 2003.

International Conferences

[C1] F. Gava, M. Guedj and F. Pommereau. A BSP algorithm for on-the-fly checking CTL* formulas on security protocols. *Parallel and Distributed Computing (PDCAT)*. IEEE. 2012. to appear.

[C2] F. Gava, A. Hidalgo and J. Fortin. Mechanised verification of distributed state-space algorithms for security protocols. *Parallel and Distributed Computing (PDCAT)*. IEEE. 2012. to appear.

[C3] F. Gava, M. Guedj and F. Pommereau. A BSP algorithm for on-the-fly checking LTL formulas on security protocols. In *International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE. 2012. to appear.

[C4] F. Gava, C. Li and G. Hains: Implementation of data-parallel skeletons: a case study using a coarsed-grained hierarchical model. In *International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE. 2012. to appear.

[C5] F. Loulergue, F. Gava, N. Kosmatov and M. Lemerre. Towards Verified Cloud Computing Environments. In *High Performance Computing and Simulation (HPCS)*. IEEE, 2012. to appear.

[C6] F. Gava, M. Guedj and F. Pommereau. Performance Evaluations of a BSP Algorithm for State Space Construction of Security Protocols. In *Parallel, Distributed, and Network-Based Processing (PDP)* 2012. pages 170-174. 2012.

[C7] S. Tan and F. Gava. Modular Implementation of Dense Matrix Operations in a High-level BSP Language. In *High Performance Computing & Simulation (HPCS)*, pages 643–649. 2010. IEEE.

[C8] J. Fortin and F. Gava. From BSP Routines to High-performance ones: Formal Verification of a Transformation Case. In *International Conference on Computational Science (ICCS)*. Procedia CS 1(1). pages 155–164, 2010.

[C9] F. Gava and J. Fortin. Two Formal Semantics of a Subset of the Paderborn University BSPlib. In D. El Baz, F. Spies, and T. Gross, editors, *Parallel, Distributed, and Network-Based Processing (PDP)*, number P3544, pages 44–51, 2009. IEEE Computer Society.

[C10] F. Gava and Jean Fortin. Formal Semantics of a Subset of the Paderborn's BSPlib. *Parallel and Distributed Computing (PDCAT)* pages 269-276. IEEE Computer Society. 2008.

[C11] F. Gava BSP Functional Programming; Examples of a cost based methodology. In M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *The International Conference on Computational Science (ICCS)*, Part I, volume 5101 of LNCS, Springer-Verlag. pages 375-385, 2008.

[C12] F. Gava Implementation of the Parallel Superposition in Bulk-Synchronous Parallel ML. In Y. Shi, G.D.v. Albada, J. Dongarra and P.M.A. Sloot, editors, *The International Conference on Computational Science (ICCS), Part IV*, volume 4487. LNCS, pages 611–619. Springer Verlag, 2007.

[C13] L. Gesbert, F. Gava, F. Loulergue and F. Dabrowski Bulk Synchronous Parallel ML with Exceptions. In P. Kacsuk, T. Fahringer and Z. Nemeth, editors, Distributed and Parallel Systems (DAPSYS), 33–42, LNCS, 2006.

[C14] F. Loulergue, R. Benheddi, F. Gava and D. Louis-Resgis In D. Grigoriev, J. Harrison and E.A. Hirsch, editors, Bulk-Synchronous Parallel ML: Semantics and Implementation of the Parallel Juxtaposition. *International Computer Science Symposium in Russia (CSR)*, 475–486, volume 3967. LNCS. 2006.

[C15] F. Loulergue, F. Gava and D. Billiet Bulk-Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam and G. Dick van Albada and P. M. A. Sloot and J. Dongarra, editors, *The International Conference on Computational Science (ICCS), Part IV*, LNCS, pages 1046–1054. Springer Verlag, 2005.

[C16] F. Gava. Parallel I/O in Bulk Synchronous Parallel ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS), Part III*, LNCS, pages 339–346. Springer Verlag, 2004.

[C17] F. Gava. Design of Deparmental Metacomputing ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS)*, LNCS, pages 50–53. Springer Verlag, 2004.

[C18] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications (ParCo)*, 2003. Elsevier, 2004.

[C19] F. Gava, F. Loulergue and F. Dabrowski. A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, pages 293–300. ACIS, 2003.

[C20] F. Dabrowski, F. Loulergue and F. Gava. Pattern Matching of Parallel Values in Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, pages 301–308. ACIS, 2003.

[C21] M. Arapinis, F. Loulergue, F. Gava and F. Dabrowski. Semantics of Minimally Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, pages 260–267. ACIS, 2003.

[C22] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT)*, volume 2763 of LNCS, pages 215–229. Springer Verlag, 2003.

[C23] F. Gava and F. Loulergue. A Parallel Virtual Machine for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS), Part I*, volume 2657 of LNCS, pages 155–164. Springer Verlag, 2003.

International Workshops

[W1] F. Gava , L. Gesbert and F. Loulergue. Type System for a Safe Execution of Parallel Programs in BSML. In 5th *SIGPLAN Workshop on High-Level Parallel Programming and Applications (HLPP, affiliated to conference ICFP)*, 2011. ACM.

[W2] F. Gava, M. Guedj and F. Pommereau. A BSP algorithm for the state space construction of security protocols. In *9th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC, affiliated to conference SPIN 2010)*, 2010. IEEE Computer Society.

[W3] W. Bousdira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot. Functional Parallel Programming with Revised Bulk Synchronous Parallel ML. In Koji Nakano, editor, *Workshop on Parallel and Distributed Algorithms and Applications (PDAA)*, 2010. IEEE Computer Society.

[W4] J. Fortin and F. Gava. BSP-Why: an intermediate language for deductive verification of BSP programs. *In SIGPLAN Workshop on High-level Parallel Programming and Applications (HLPP, affiliated to conference ICFP)*, 2010. ACM Press.

[W5] F. Gava and I. Garnier. New Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons. In *Workshop APDCM, part of IPDPS*, pages 1–8, 2009. IEEE Computer Society.

National Journal

[A1] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. *Techniques et sciences informatiques*, 25(10):(1261–1280), 2007. Extended version of [N4].

National Conferences

[N1] F. Gava and S. Tan. Implementation et prédiction des performances de squelettes data-parallèles en utilisant un langage BSP de haut-niveau. In S. Conchon and A. Mahboudi, editors, *Journées Francophones des Langages Applicatifs (JFLA)*, Studia Informatica Universalis, pages 39–65, 2011. Hermann.

[N2] L. Gesbert, F. Gava, F. Loulergue and F. Dabrowski Bulk Synchronous Parallel ML avec exceptions. Rencontres Francophones du Parallélisme (Renpar), 2006.

[N3] F. Gava. Une implantation de la juxtaposition. In T. Hardin, editor, *Journées Francophones des Langages Applicatifs (JFLA)*, pages 87–100, INRIA, january 2006,.

[N4] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In V. Ménissier Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA)*, pages 55–68, INRIA, 2004.

[N5] F. Gava and F. Loulergue. Synthèse de types pour Bulk Synchronous Parallel ML. In J.-C. Filliâtre, editor, *Journées Francophones des Langages Applicatifs (JFLA)*, pages 153–168, INRIA, 2003.

Thesis

[M1] F. Gava. Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs ; Sémantiques, implantations et certification. Doctoral thesis, University of Paris XII–Val-de-Marne, 2005.

[M2] F. Gava. Un système de type polymorphe pour le langage BSML avec traits impératifs. Master's thesis, University of Paris XII–Val-de-Marne, 2002.

Bibliography

[1] C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4), 2009. Page 5.

[2] B. Hendrickson. Computational science: Emerging opportunities and challenges. *Journal of Physics: Conference Series*, 180(1), 2009. Pages 5 and 10.

[3] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, and Jesper Larsson Träff. Mpi on millions of cores. *Parallel Processing Letters*, 21(1):45–60, 2011. Pages 5 and 95.

[4] Cliff B. Jones, Peter W. O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006. Page 5.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006. Page 5.

[6] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004. Pages 5, 7, 9 and 93.

[7] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006. Pages 6, 7, 9, 63 and 93.

[8] Kevin Hammond and Greg Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 2000. Pages 6 and 42.

[9] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel hpc applications. In *Computer Communications and Networks (ICCCN)*, pages 1–8. IEEE, 2010. Pages 7, 42, 48 and 97.

[10] 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. Pages 7, 12, 67, 68 and 95.

[11] Shin'ichiro Matsuo, Kunihiko Miyazaki, Akira Otsuka, and David A. Basin. How to evaluate the security of real-life cryptographic protocols? - the cases of iso/iec 29128 and cryptrec. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, Josep M. Miret, Kazue Sako, and Francesc Sebé, editors, *Financial Cryptography and Data Security Workshops*, volume 6054 of *LNCS*, pages 182–194. Springer, 2010. Page 7.

[12] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. Pages 7 and 68.

[13] 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 7.

[14] 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 7.

[15] Catherine Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, 2003. Page 7.

[16] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 46(3-4):225–259, 2011. Pages 7 and 87.

[17] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Pages 7, 68 and 87.

[18] Alessandro Armando, Roberto Carbone, and Luca Compagna. Ltl model checking for security protocols. *Journal of Applied Non-Classical Logics*, 19(4):403–429, 2009. Page 7.

[19] H. Gao. *Analysis of Security Protocols by Annotations*. PhD thesis, Technical University of Denmark, 2008. Pages 7, 68 and 89.

[20] 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 7, 75 and 89.

[21] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In *ENTCS*, volume 128, pages 19–34. Elsevier, 2005. Page 8.

[22] B. Maggs, L. Matheson, and R. Tarjan. Models of parallel computation: A survey and synthesis. In *Proceedings of the 28th Annual Hawaii International Conference on Systems Sciences*, 1995. Motivates explicit processes and the two-level memory hierarchy of BSP and LogP. Page 8.

[23] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990. Page 8.

[24] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. Pages 8 and 9.

[25] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004. Pages 8, 9, 17, 18, 22, 42, 58 and 96.

[26] Thilo Kielmann and Sergei Gorlatch. Bandwidth-latency models (bsp, logp). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 107–112. Springer, 2011. Page 8.

[27] 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 8.

[28] 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 8 and 42.

[29] A. N. Yzelman and Rob H. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, 2012. Pages 8 and 42.

[30] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: bulk-synchronous gpu programming. *ACM Trans. Graph.*, 27(3), 2008. Pages 8, 10 and 42.

[31] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999. Pages 9 and 96.

[32] M. Snir and W. Gropp. *MPI the Complete Reference.* MIT Press, 1998. Page 9.

[33] M. Armbrust, A. Fox, R. Griffith, and al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. Page 9.

[34] Jörn Eisenbiegler, Welf Löwe, and Wolf Zimmermann. BSP, LogP, and oblivious programs. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, pages 865–874, London, UK, 1998. Springer-Verlag. Page 9.

[35] 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 9.

[36] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP versus LogP. *Algorithmica*, 24:405–422, 1999. Page 9.

[37] 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 9.

[38] D. R. Martinez, J. C. Cabaleiro, T. F. Pena, F. F. Rivera, and V. Blanco. Accurate analytical performance model of communications in MPI applications. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009. Page 9.

[39] Chee-Kong Chui. The logp and mlogp models for parallel image processing with multi-core microprocessor. In *Symposium on Information and Communication Technology (SoICT)*, pages 23–27. ACM, 2010. Page 9.

[40] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10:127–143, June 2007. Page 9.

[41] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994. Pages 9 and 18.

[42] 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. Pages 9 and 10.

[43] 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 9.

[44] 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 9.

[45] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms.* PhD thesis, Oxford University Computing Laboratory, 1998. Pages 9, 10, 19, 20, 58 and 63.

[46] A.V.Gerbessiotis. *Topics in Parallel and Distributed Computation.* PhD thesis, Harvard University, 1993. Page 9.

[47] I. Gu'erin-Lassous and J. Gustedt. Portable List Ranking: an Experimental Study. *ACM Journal of Experiments Algorithms*, 7(7):1–18, 2002. Page 9.

[48] 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 9.

[49] 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 9.

[50] 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 9.

[51] 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 9.

[52] M. Bamha and G. Hains. Frequency-adaptive join for Shared Nothing machines. *Parallel and Distributed Computing Practices*, 2(3):333–345, 1999. Page 9.

[53] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003. Page 9.

[54] 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 9.

[55] 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 10.

[56] 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. Pages 10 and 42.

[57] I. Guerin Lassous. *Algorithmes paralleles de traitement de graphes: une approche basee sur l'analyse experimentale*. PhD thesis, University de Paris VII, 1999. Page 10.

[58] 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 10.

[59] 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 10.

[60] P Ferragina and F. Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24(3):177–194, 1999. Page 10.

[61] 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 10.

[62] 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 10.

[63] Veronica Gil Costa and Mauricio Marín. A parallel search engine with bsp. In *Latin American Web Congress (LA-WEB)*, pages 259–268. IEEE Computer Society, 2005. Page 10.

[64] Olaf Bonorden. *Versatility of bulk synchronous parallel computing: from the heterogeneous cluster to the system on chip*. PhD thesis, University of Paderborn, 2008. Page 10.

[65] Olaf Bonorden, Joachim von zur Gathen, Jurgen Gerhard, Olaf Muller, and Michael Nocker. Factoring a binary polynomial of degree over one million. *ACM SIGSAM Bulletin*, 35(1):16–18, 2001. Page 10.

[66] Fatima Khaled Abu Salem. *Factorisation Algorithms for Univariate and Bivariate Polynomials over Finite Fields*. PhD thesis, University of Oxford (Merton College), 2004. Page 10.

[67] Fatima K. Abu Salem and Laurence Tianruo Yang. Parallel methods for absolute irreducibility testing. *The Journal of Supercomputing*, 46(3):181–212, 2008. Page 10.

[68] Radu Calinescu. Bulk synchronous parallel algorithms for optimistic discrete event simulation. Technical Report PRG-TR-8-96, Programming Research Group, Oxford University Computing Laboratory, April 1996. Page 10.

[69] Chun-Hsi Huang and Sanguthevar Rajasekaran. High-performance parallel bio-computing. *Parallel Computing*, 30(9-10):999–1000, 2004. Page 10.

[70] Rodrigo da Rosa Righi, Laércio Lima Pilla, Nicolas Maillard, Alexandre Carissimi, and Philippe Olivier Alexandre Navaux. Observing the impact of multiple metrics and runtime adaptations on bsp process rescheduling. *Parallel Processing Letters*, 20(2):123–144, 2010. Page 10.

[71] Bin Cheng, Yan Jiang, and Weiqin Tong. Hierarchical resource load balancing based on multi-agent in servicebsp model. *International Journal of Communications, Network and System Sciences (IJCNS)*, 3(1):59–65, 2010. Page 10.

[72] Rodrigo da Rosa Righi, Laércio Lima Pilla, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, and Hans-Ulrich Heiss. Migbsp: A novel migration model for bulk-synchronous parallel processes rescheduling. In *Conference on High Performance Computing and Communications (HPCC)*, pages 585–590. IEEE, 2009. Pages 10 and 42.

[73] L. Gesbert. *Développement systématique et sûreté d'exécution en programmation parallèle structurée*. PhD thesis, University Paris Est, LACL, 2009. Pages 11, 13, 15, 24, 28, 29, 31, 34, 41, 45 and 93.

[74] Michael Guedj. *BSP Algorithms for LTL & CTL* Model Checking of Security Protocols*. PhD thesis, University of Paris-Est, 2012. Pages 11, 70, 81, 84 and 85.

[75] 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. Page 12.

[76] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. *Applied Non-Classical Logics*, 19(4):403–429, 2009. Pages 12, 68, 84, 86, 87, 88 and 97.

[77] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. Pages 14, 18, 36, 43 and 93.

[78] 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. Pages 14, 42 and 49.

[79] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In J. Ramanujam and P. Sadayappan, editors, *Principles and Practice of Parallel Programming (PPOPP)*, pages 181–192. ACM, 2012. Page 14.

[80] F. Loulergue, G. Hains, and C. Foisy. A calculus of functional BSP programs. *Science of Computer Programming*, 37(1–3):253–277, May 2000. Pages 14 and 23.

[81] Frédéric Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, 11(4):423–437, 2001. Pages 14 and 15.

[82] O. Ballereau, F. Loulergue, and G. Hains. High level BSP programming: BSML and BSλ. In G. Michaelson, P. Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming*, chapter 4, pages 29–38. Intellect Books, Bristol UK, Portland, OR, USA, August 2000. Preliminary version appeared in proceedings of SFP'99: First Scottish Functional Programming Workshop, TR RM/99/9, pp.43–52, Heriot-Watt University, September 1999. Page 14.

[83] A. Merlin, G. Hains, and F. Loulergue. An SPMD environment machine for functional BSP programs. In *SFP'2001 Scottish Functional Programming Workshop*, Stirling, August 2001. Page 14.

[84] Frédéric Loulergue. Implementation of a functional bulk synchronous parallel programming library. In Selim G. Akl and Teofilo F. Gonzalez, editors, *IASTED Parallel and Distributed Computing Systems (PDCS)*, pages 447–452. IASTED/ACTA Press, 2002. Page 14.

[85] Frédéric Dabrowski and Frédéric Loulergue. Functional Bulk Synchronous Programming in C++. In 21*st IASTED International Multi-conference, Applied Informatics (AI 2003), Symposium on Parallel and Distributed Computing and Networks*, pages 462–467. ACTA Press, february 2003. Page 14.

[86] Louis Gesbert, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, and Julien Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE Computer Society, 2010. Pages 15 and 41.

[87] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. Page 15.

[88] Julien Tesson. *Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels*. PhD thesis, LIFO, University of Orléans, November 2011. Pages 15 and 41.

[89] Frédéric Loulergue. Parallel composition and bulk synchronous parallel functional programming. In Stephen Gilmore, editor, *Scottish Functional Programming Workshop*, volume 2 of *Trends in Functional Programming*, pages 77–88. Intellect, 2000. Page 15.

[90] Frédéric Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS)*, volume 2659 of *LNCS*, pages 223–232. Springer, 2003. Pages 15 and 24.

[91] O. Bonorden, B. Judoiink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003. Pages 16, 17, 42 and 49.

[92] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, 2007. Pages 18, 38 and 39.

[93] A. V. Gerbessiotis. Algorithmic and practical considerations for dense matrix computations on the bsp model. Technical Report PRG-TR-32-97, The University of Oxford, 1997. Page 22.

[94] J. M. Del Rosario and A. Choudhary. High performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, 1994. Page 24.

[95] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36:97–122, 2003. Pages 24 and 45.

[96] A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, (4), 2001. Page 24.

[97] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. Page 25.

[98] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Second ACM SIGPLAN Workshop on Continuations*, 1997. Page 25.

[99] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, 1994. Page 26.

[100] J. Kim and K. Yi. Interconnecting between CPS terms and non-CPS terms. In A. Sabry, editor, *SIGPLAN Workshop on Continuations*. ACM, 2001. Page 26.

[101] N. Heintze. Control-Flow Analysis and Type Systems. In A. Mycroft, editor, *Static Analysis Symposium (SAS)*, number 983 in LNCS. Springer, 1995. Page 26.

[102] O. Danvy and L. R. Nielsen. Cps transformation of beta-redexes. *Information Processing Letterseses*, 94(5):217–225, 2005. Page 27.

[103] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR*, volume 4790 of *LNAI*, pages 211–225. Springer, 2007. Page 27.

[104] J. H. Reif. Depth-first search is inherrently sequential. *Information Processing Letters*, 20(5):229–234, 1985. Page 29.

[105] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000. Page 32.

[106] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, June 1994. Page 32.

[107] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Page 32.

[108] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. Page 34.

[109] Joel Falcou. Parallel programming with skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009. Page 36.

[110] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software, Practrice & Experience*, 40(12):1135–1160, 2010. Pages 36 and 43.

[111] Sergei Gorlatch and Murray Cole. Parallel skeletons. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer, 2011. Page 36.

[112] S. Gorlatch. SAT: A Programming Methodology with Skeletons and Collective Operations. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 29–64. Springer, 2003. Pages 36 and 43.

[113] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002. Pages 36, 43 and 98.

[114] R. Di Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3L 2.0. *Parallel Processing Letters*, 2008. to appear. Page 36.

[115] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: a structured high-level parallel language, and its structure support. *Concurrency: Practice and Experiences*, 7(3):225–255, May 1995. Page 36.

[116] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP3l. *Parallel Computing*, 32:539–550, 2006. Pages 36, 37 and 43.

[117] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. Pages 40, 59, 65, 66, 71 and 93.

[118] Julien Tesson and Frédéric Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *11th International Conference on Computational Science (ICCS 2011)*, Procedia Computer Science, pages 36–45. Elsevier, 2011. Page 41.

[119] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, 32(7–8):590–603, 2006. Pages 41 and 43.

[120] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale'06: Proceedings of the 1st international conference on Scalable information systems*. ACM Press, 2006. Page 41.

[121] Wadoud Bousdira, Frédéric Loulergue, and Julien Tesson. A verified library of algorithmic skeletons on evenly distributed arrays. In Yang Xiang, Ivan Stojmenovic, Bernady O. Apduhan, Guojun Wang, Koji Nakano, and Albert Y. Zomaya, editors, *Algorithms and Architectures for Parallel Processing (ICA3PP)*, volume 7439 of *LNCS*, pages 218–232. Springer, 2012. Page 41.

[122] Noman Javed and Frédéric Loulergue. Verification of a heat diffusion simulation written with orléans skeleton library. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics (PPAM)*, volume 7204 of *LNCS*, pages 91–100. Springer, 2011. Page 41.

[123] Kosuke Ono, Yoichi Hirai, Yoshinori Tanabe, Natsuko Noda, and Masami Hagiya. Using coq in specification and program extraction of hadoop mapreduce applications. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods (SEFM)*, volume 7041 of *LNCS*, pages 350–365. Springer, 2011. Page 41.

[124] Yu Liu, Zhenjiang Hu, and Kiminori Matsuzaki. Towards systematic parallel programming over mapreduce. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Parallel Processing (Euro-Par)*, volume 6853 of *LNCS*, pages 39–50. Springer, 2011. Page 41.

[125] B. Di Martino, A. Mazzeo, M. Mazzocca, and U. Villano. Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. *Science of Computer Programming*, 40(2–3):235–263, 2001. Pages 42, 65 and 97.

[126] O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. A Web Computing Environment for Parallel Algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006. Page 42.

[127] Gregory F. Diamos, Andrew R. Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Parallel architectures and compilation techniques (PACT)*, pages 353–364. ACM, 2010. Page 42.

[128] Y. Gu, B.-S. Le, and C. Wentong. JBSP: a BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001. Page 42.

[129] Lucas Graebin and Rodrigo da Rosa Righi. jmigbsp: Object migration and asynchronous one-sided communication for bsp applications. In Sang-Soo Yeo, Binod Vaidya, and George A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 35–38. IEEE Computer Society, 2011. Page 42.

[130] Sangwon Seo, Edward J. Yoon, Jae-Hong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing (CloudCom)*, pages 721–726. IEEE, 2010. Pages 42, 49 and 96.

[131] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010. Pages 42 and 43.

[132] Matthew Felice Pace. Bsp vs mapreduce. *Procedia CS*, 9:246–255, 2012. Page 42.

[133] Christoph W. Keßler. Neststep: Nested parallelism and virtual shared memory for the bsp model. *The Journal of Supercomputing*, 17(3):245–262, 2000. Page 42.

[134] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. A framework for an automatic hybrid mpi+openmp code generation. In Layne T. Watson, Gary W. Howell, William I. Thacker, and Steven Seidel, editors, *Simulation Multiconference (SpringSim) on High Performance Computing Symposia (HPC)*, pages 48–55. SCS/ACM, 2011. Pages 42 and 96.

[135] Konrad Hinsen, Hans Petter Langtangen, Ola Skavhaug, and Åsmund Ødegård. Using BSP and Python to simplify parallel programming. *Future Generation Comp. Syst.*, 22(1-2):123–157, 2006. Page 42.

[136] Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002. Page 42.

[137] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher Order and Symb. Comp.*, 16(3):203–251, 2003. Pages 42, 43 and 45.

[138] Kevin Hammond. Parallel functional programming: An introduction. In *International Symposium on Parallel Symbolic Computation*. World Scientific, 1994. Page 42.

[139] Guy E. Blelloch. Nesl. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1278–1283. Springer, 2011. Page 42.

[140] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *POPL's Workshop Declarative Aspects of Multicore Programming (DAMP)*. ACM, 2007. Page 42.

[141] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in manticore. *SIGPLAN Not.*, 43(9):119–130, 2008. Page 43.

[142] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. Page 43.

[143] C. Grelck and Sven-Bodo Scholz. Classes and objects as basis for I/O in SAC. In *Proceedings of IFL'95*, pages 30–44, Gothenburg, Sweden, 1995. Page 43.

[144] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In Neal Glew and Guy E. Blelloch, editors, *POPL's workshop Declarative Aspects of Multicore Programming (DAMP)*, pages 10–18. ACM, 2007. Page 43.

[145] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari. Parallel functional programming in eden. *Journal of Functional Programming*, 3(15):431–475, 2005. Pages 43 and 45.

[146] Kevin Hammond. Glasgow parallel haskell (gph). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 768–779. Springer, 2011. Pages 43 and 45.

[147] Jost Berthold, Mischa Dieterle, Oleg Lobachev, and Rita Loogen. Parallel FFT with Eden Skeletons. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 5698 of *LNCS*, pages 73–83. Springer, 2009. Page 43.

[148] N. Scaife, G. Michaelson, and Susumu Horiguchi. Empirical Parallel Performance Prediction From Semantics-Based Profiling. *Scalable Computing: Practice and Experience*, 7(3), 2006. Page 43.

[149] Kevin Hammond. The dynamic properties of hume: A functionally-based concurrent language with bounded time and space behaviour. In Markus Mohnen and Pieter W. M. Koopman, editors, *Implementation of Functional Languages (IFL)*, volume 2011 of *LNCS*, pages 122–139. Springer, 2000. Page 43.

[150] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003. Page 43.

[151] Herbert Kuchen and J. Striegnitz. Features from functional programming for a c++ skeleton library. *Concurrency - Practice and Experience*, 17(7-8):739–756, 2005. Page 43.

[152] M. Aldinucci and M. Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, 2007. Page 43.

[153] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, 2006. Pages 43 and 98.

[154] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003. Page 43.

[155] Martin Alt, Holger Bischof, and Sergei Gorlatch. Algorithm design and performance prediction in a java-based grid system with skeletons. In *EUROPAR*, volume 2790 of *LNCS*, pages 899–906. Springer, 2003. Page 43.

[156] Mario Leyton and José M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *PDP*, pages 289–296. IEEE, 2010. Page 43.

[157] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: An expandable skeleton environment. *Scientific International Journal for Parallel and Distributed Computing*, 8:325–341, 2007. Page 43.

[158] P. Ciechanowicz and H. Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *HPCC*, pages 108–113. IEEE, 2010. Page 43.

[159] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 45–53. IEEE Computer Society, 2008. Page 43.

[160] C. A. Herrmann and C. Lengauer. Hdc: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000. Page 43.

[161] A. Zavanella. Skeletons, bsp and performance portability. *Parallel Processing Letters*, 11(4):393–407, 2001. Page 43.

[162] Y. Hayashi and M. Cole. Automated bsp cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters*, 12(1):95–112, 2002. Page 43.

[163] F. O. Osoba and F. A. Rabbi. A parallel multigrid skeleton using BSP. *LNCS*, 1470:704–??, 1998. Page 43.

[164] J. Sérot and J. Falcou. Functional meta-programming for parallel skeletons. In *8th International Conference Computational Science (ICCS)*, volume 5101 of *LNCS*, pages 154–163. Springer, 2008. Page 43.

[165] J. Serot, D. Ginhac, R. Chapuis, and J.P. Derutin. Fast prototyping of parallel vision applications using functional skeletons. *Journal of Mahcine Vision and Applications*, 12(6):271–290, 2001. Page 43.

[166] Tore A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *5th International Workshop on the Implementation of Functional Languages*, pages 23–33, Numegen, The Netherlands, September, 1993. Page 43.

[167] Norman Scaife, Susumu Horiguchi, Greg Michaelson, and Paul Bristow. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming*, 15(4):615–650, 2005. Page 43.

[168] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization with tree skeletons. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *LNCS*, pages 789–798. Springer, 2003. Page 43.

[169] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming (ESOP)*, number 2305 in LNCS, pages 83–97. Springer, 2002. Page 43.

[170] Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin, editors, *Euro-Par*, volume 5704 of *Lecture Notes in Computer Science*, pages 666–677. Springer, 2009. Page 43.

[171] Y. Karasawa and H. Iwasaki. A parallel skeleton library for multi-core clusters. In *ICPP*, pages 84–91. IEEE Computer Society, 2009. Page 43.

[172] Mario Leyton, Ludovic Henrio, and José M. Piquer. Exceptions for algorithmic skeletons. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *16th International Euro-Par Conference*, LNCS 6272, pages 14–25. Springer, 2010. Page 44.

[173] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992. Page 44.

[174] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989. Page 44.

[175] John Feo, David C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990. Page 44.

[176] Nicholas Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. The linda® alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994. Page 44.

[177] Ian T. Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The pcn approach. *Scientific Programming*, 1(1):51–66, 1992. Page 44.

[178] L. Mandel and L. Maranget. Programming in JoCaml. In *European Symposium on Programming (ESOP)*, LNCS. Springer-Verlag, 2008. Page 44.

[179] Andreas Rossberg. *Typed Open Programming – A higher-order, typed approach to dynamic modularity and distribution*. PhD thesis, Universität des Saarlandes, 2007. Page 44.

[180] Gert Smolka. The oz programming model. In *Computer Science Today*, LNCS, pages 324–343. Springer-Verlag, 1995. Page 44.

[181] D. E. Culler, A. Dusseau, S. C. Goldstein, and A. Krishnamurthy. Parallel programming in Split-C. In *Supercomputing'93*, November 1993. Page 44.

[182] Tarek A. El-Ghazawi and Lauren Smith. Upc: unified parallel c. In *High Performance Networking and Computing (SC)*, page 27. ACM Press, 2006. Page 44.

[183] Bradford L. Chamberlain. Zpl. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 2161–2166. Springer, 2011. Page 44.

[184] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992. Page 44.

[185] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Fifth ACM-SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 1995. Page 44.

[186] Laxmikant Kale. Charm++. In David A. Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011. Page 44.

[187] Stephan Herhut, Sven-Bodo Scholz, and Clemens Grelck. Controlling chaos: on safe side-effects in data-parallel operations. In Leaf Petersen and Manuel M. T. Chakravarty, editors, *POPL Workshop on Declarative Aspects of Multicore Programming (DAMP)*, pages 59–67. ACM, 2009. Page 45.

[188] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory, two -level memories. *Algorithmica*, 12(2):110–147, 1994. Page 45.

[189] A. Gordon and R. L. Crole. A sound metalogical semantics for input/output effects. *Mathematical Structures in Computer Science*, 9:125–188, 1999. Page 45.

[190] M. Dowse and A. Butterfield. A language for reasoning about concurrent functional I/O. Technical Report 0408, Institut fur Informatik, Lubeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds. Page 45.

[191] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 889–890, Baltimore, MD, 1999. Page 45.

[192] J. Gustedt. Towards realistic implementations of external memory algorithms using a coarse grained paradigm. Technical Report 4719, INRIA, 2003. Page 45.

[193] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *18th International Conference, on Term Rewriting and Applications*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007. Page 45.

[194] J.-L. Giavitto and O. Michel. Pattern-matching and rewriting rules for group indexed data structures. In B. Fischer and E. Visser, editors, *ACM SIGPLAN Workshop on Rule-Based Programming*, pages 55–66. ACM, 2002. Page 45.

[195] Alexander Romanovsky and Jörg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Advances in exception handling techniques*, pages 147–164. Springer, 2001. Page 45.

[196] A. B. Romanovsky, C. Dony, J. Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*. Springer, 2001. Page 45.

[197] Simon Marlow, Simon Peyton Jones, and Andrew Moran. Asynchronous exceptions in haskell. In *Programming Languages Design and Implementation (PLDI)*, pages 274–285. ACM, 2001. Page 45.

[198] Henri E. Bal. Fault-tolerant parallel programming in argus. *Concurrency: Pract. Exper.*, 4(1):37–55, 1992. Page 46.

[199] J. D. Harrop. Objective Caml for Scientists, 2005. Page 46.

[200] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of mpi-based parallel programs. *Commun. ACM*, 54(12):82–91, 2011. Pages 48 and 64.

[201] G. Gopalakrishnan and R. M. Kirby. Runtime verification methods for MPI. In *IPDPS*, pages 1–5. IEEE, 2008. Page 48.

[202] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 51. IEEE Computer Society, 2000. Page 48.

[203] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003. Page 48.

[204] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel®Message Checker. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82. ACM, 2005. Page 48.

[205] I. Grudenic and N. Bogunovic. Modeling and Verification of MPI Based Distributed Software. In B. Mohr, J. Larsson Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, LNCS 4192, pages 123–132. Springer, 2006. Page 48.

[206] P. Ohly and W. Krotz-Vogel. Automated MPI Correctness Checking: What if There Were a Magic Option? In *The 8th LCI International Conference on High-Performance Clustered Computing*, California, USA, 2007. Page 48.

[207] S. F. Siegel and G. S. Avrunin. Verification of Halting Properties for MPI Programs Using Nonblocking Operations. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2007)*, LNCS 4757, pages 326–334. Springer, 2007. Page 48.

[208] G. Li, M. Delisi, G. Gopalakrishnan, and R. M. Kirby. Formal specification of the MPI-2.0 standard in TLA+. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 283–284. ACM, 2008. Page 48.

[209] B. Krammer and M. M. Resch. Correctness Checking of MPI One-Sided Communication Using Marmot. In B. Mohr, J. Larsson Träff, J. Worringen, and J. Dongarra, editors, *Euro-PVM/MPI*, volume 4192 of *LNCS*, pages 105–114. Springer, 2006. Page 48.

[210] Stephen F. Siegel. Verifying parallel programs with MPI-Spin. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Euro PVM/MPI*, volume 4757 of *LNCS*, pages 13–14. Springer, 2007. Pages 48, 64 and 97.

[211] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal Verification of Practical MPI Programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 261–269, 2009. Pages 48, 64 and 97.

[212] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a tool for model checking MPI programs. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 285–286. ACM, 2008. Page 48.

[213] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009. Page 48.

[214] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. Page 49.

[215] F.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4), 2003. Page 49.

[216] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS. Springer, 2007. Page 49.

[217] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, 2012. Page 49.

[218] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *International Workshop on Intermediate Verification Languages (Boogie)*, 2011. Page 49.

[219] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004. `http://why.lri.fr/caduceus/`. Page 49.

[220] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS. Springer-Verlag, 2007. Pages 49 and 96.

[221] Jérôme Guitton, Johannes Kanig, and Yannick Moy. Why hi-lite ada? In *Boogie Workshop*, 2011. Page 49.

[222] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364, 2005. Page 51.

[223] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990. Page 59.

[224] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. Page 59.

[225] X. Leroy and H. Grall. Coinductive Big-step Operational Semantics. *Information and Computation*, 2008. to appear. Page 60.

[226] J. Tesson and F. Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPlib Library. In J. Weglarz, R. Wyrzykowski, and B. Szymanski, editors, *Parallel Processing and Applied Mathematics (PPAM)*, number 4967 in LNCS, pages 1122–1129. Springer, 2007. Page 61.

[227] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 5–21. Springer, 2007. Page 61.

[228] J. Kanig and J.-C. Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, 2009. Page 62.

[229] Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC)*, pages 305–335, 2008. Page 62.

[230] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the spec# experience. *Commun. ACM*, 54(6):81–91, 2011. Pages 62 and 72.

[231] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008. Page 62.

[232] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976. Page 62.

[233] Alexander Malkis and Laurent Mauborgne. On the strength of owicki-gries for resources. In Hongseok Yang, editor, *Programming Languages and Systems (APLAS)*, volume 7078 of *LNCS*, pages 172–187. Springer, 2011. Page 63.

[234] Tobias Nipkow and Leonor Prensa Nieto. Owicki/gries in isabelle/hol. In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS*, volume 1577 of *LNCS*, pages 188–203. Springer, 1999. Page 63.

[235] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Institut fur Informatik, Technische Universitat Munchen, 2001. Pages 63 and 65.

[236] Leonor Prensa Nieto and Javier Esparza. Verifying single and multi-mutator garbage collectors with owicki-gries in isabelle/hol. In Mogens Nielsen and Branislav Rovan, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of *LNCS*, pages 619–628. Springer, 2000. Page 63.

[237] Rob Gerth and Willem P. de Roever. A proof system for concurrent ada programs. *Sci. Comput. Program.*, 4(2):159–204, 1984. Page 63.

[238] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007. Page 63.

[239] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *17th European Symposium on Programming (ESOP)*, number 4960 in LNCS, pages 353–367. Springer, 2008. Page 63.

[240] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007. Page 63.

[241] François Bobot and Jean-Christophe Filliâtre. Separation predicates: a taste of separation logic in first-order logic. In *Formal Ingineering Methods (ICFEM)*, volume 7635 of *LNCS*. Springer, 2012. Page 63.

[242] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002. Page 63.

[243] K. Rustan M. Leino and Peter Muller. A basis for verifying multi-threaded programs. In *ESOP*, LNCS. Springer, 2009. Page 63.

[244] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *ICFEM*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006. Page 63.

[245] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Page 63.

[246] K. Rustan M. Leino. Verifying concurrent programs with chalice. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of *LNCS*, page 2. Springer, 2010. Page 63.

[247] Leslie Lamport and Fred B. Schneider. The "hoare logic" of csp, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296, 1984. Page 64.

[248] Matthias Daum. Reasoning on data-parallel programs in isabelle/hol, 2007. Page 64.

[249] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. Formal Validation of Data–Parallel Programs : a Two–Component Assertional Proof System for a Simple Language. *Theoretical Computer Science*, 189(1-2):71–107, 1997. Page 64.

[250] G. Li, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Formal Specification of the MPI-2.0 Standard in TLA+. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 283–284, 2008. Page 64.

[251] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced execution semantics of MPI: From theory to practice. In *FM 2009*, 2009. Page 64.

[252] S. V. Sharma, G. Gopalakrishnan, and R. M. Kirby. A survey of MPI related debuggers and tools. Technical Report UUCS-07-015, University of Utah, School of Computing, 2007. `http://www.cs.utah.edu/research/techreports.shtml`. Page 64.

[253] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel-Message Checker. In *SE-HPCS'05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82. ACM, 2005. Page 64.

[254] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003. Page 64.

[255] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000. Page 64.

[256] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing*, 2003. Page 64.

[257] B. Krammer, M. S. Müller, and M. M. Resch. MPI I/O Analysis and Error Detection with MARMOT. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *Euro PVM/MPI*, volume 3241 of *LNCS*, pages 242–250. Springer, 2004. Page 64.

[258] J. L. Traeff and J. Worringen. Verifying Collective MPI Calls. In *Proceedings of the 11th EuroPVM/MPI conference*, LNCS. Springer, 2004. Page 64.

[259] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Formal verification of programs that use MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, volume 4192 of *LNCS*, pages 30–39. Springer, 2006. Page 64.

[260] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, volume 5205 of *LNCS*, pages 265–273. Springer, 2008. Page 64.

[261] S. F. Siegel, A. M., G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008. Page 64.

[262] Stephen F. Siegel and Timothy K. Zirkel. Tass: The toolkit for accurate scientific software. *Mathematics in Computer Science*, 5(4):395–426, 2011. Page 64.

[263] Stephen F. Siegel and Timothy K. Zirkel. Fevs: A functional equivalence verification suite for high-performance scientific computing. *Mathematics in Computer Science*, 5(4):427–435, 2011. Page 64.

[264] Stephen F. Siegel and Timothy K. Zirkel. Loop invariant symbolic execution for parallel programs. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7148 of *LNCS*, pages 412–427. Springer, 2012. Page 64.

[265] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation*, volume 6300 of *LNCS*, pages 277–300. Springer, 2010. Pages 65 and 66.

[266] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008. Pages 65 and 66.

[267] Jeffrey M. Squyres. Zen and the art of MPI collectives. *ClusterWorld Magazine, MPI Mechanic Column*, 2(3):32–34, 2004. Page 65.

[268] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In Zhenjiang Hu, editor, *Programming Languages and Systems (APLAS)*, volume 5904 of *LNCS*, pages 194–209. Springer, 2009. Page 65.

[269] Jules Villard. *Heaps and Hops*. PhD thesis, École Normale Supérieure de Cachan, LSV, France, 2011. Page 65.

[270] D. S. Lecomber. A semantics for parallel programming with bsp. Page 65.

[271] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. *J. Funct. Program.*, 18(5-6):599–647, 2008. Page 65.

[272] Paolo Herms. Certification of a chain for deductive program verification. In Yves Bertot, editor, *2nd Coq Workshop, satellite of ITP'10*, 2010. Pages 65, 66, 72 and 73.

[273] Y. Chen and J.W. Sanders. Logic of global synchrony. *ACM Transactions on Programming Languages and Systems*, 26(2):221–262, 2004. Page 65.

[274] J. Zhou and Y. Chen. Generating c code from logs specifications. In D. Van Hung and M. Wirsin, editors, *ICTAC*, volume 3722 of *LNCS*, pages 195–210. Springer, 2005. Page 65.

[275] Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3):389–400, 2003. Page 65.

[276] H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par'96*, number 1124 in LNCS, pages 359–368. Springer, 1996. Page 65.

[277] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14:271–292, 2000. Page 65.

[278] Alan Stewart, Maurice Clint, and Joaquim Gabarró. Barrier synchronisation: Axiomatisation and relaxation. *Formal Asp. Comput.*, 16(1):36–50, 2004. Page 65.

[279] Alan Stewart. A programming model for bsp with partitioned synchronisation. *Formal Asp. Comput.*, 23(4):421–432, 2011. Page 65.

[280] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. *Commun. ACM*, 53(6):97–105, 2010. Page 65.

[281] Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012. Page 65.

[282] Alexander Malkis and Anindya Banerjee. Verification of software barriers. In J. Ramanujam and P. Sadayappan, editors, *SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 313–314. ACM, 2012. Page 65.

[283] 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 7.

[284] Steve Kremer, Olivier Markowitch, and Jianying Zhou. An intensive survey of fair non-repudiation protocols. *Computer Communications*, 25(17):1606–1621, 2002. Pages 68 and 88.

[285] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols: A taster. In David Chaum, Markus Jakobsson, Ronald L. Rivest, Peter Y. A. Ryan, Josh Benaloh, Miroslaw Kutylowski, and Ben Adida, editors, *Towards Trustworthy Elections, New Directions in Electronic Voting*, volume 6000 of *LNCS*, pages 289–309. Springer, 2010. Pages 68 and 97.

[286] Miyako Ohkubo, Fumiaki Miura, Masayuki Abe, Atsushi Fujioka, and Tatsuaki Okamoto. An improvement on a practical secret voting scheme. In Masahiro Mambo and Yuliang Zheng, editors, *International Workshop on Information Security (ISW)*, volume 1729 of *LNCS*, pages 225–234. Springer, 1999. Pages 68 and 97.

[287] Panagiotis Katsaros, Vasilis Odontidis, and Maria Gousidou-Koutita. Colored petri net based model checking and failure analysis for e-commerce protocols. In *Dept. of Computer Science, University of Aarhus*, pages 267–283, 2005. Pages 68 and 97.

[288] 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. Pages 68 and 77.

[289] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–520, 2003. Page 68.

[290] Cédric Fournet, Karthikeyan Bhargavan, and Andrew D. Gordon. Cryptographic verification by typing for a sample protocol implementation. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design (FOSAD)*, volume 6858 of *LNCS*, pages 66–100. Springer, 2011. Page 68.

[291] S. Meier, C. J. F. Cremers, and D. A. Basin. Strong invariants for the efficient construction of machine-checked protocol security proofs. In *Computer Security Foundations (CSF)*, pages 231–245. IEEE Computer Society, 2010. Pages 69 and 90.

[292] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Computer-aided cryptographic proofs. In Antoine Miné and David Schmidt, editors, *International Symposium on Static Analysis (SAS)*, volume 7460 of *LNCS*, pages 1–2. Springer, 2012. Pages 69, 72 and 90.

[293] 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. Pages 69 and 88.

[294] 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 69, 87 and 88.

[295] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006. Pages 69 and 87.

[296] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE CSFW'01*. IEEE Computer Society, 2001. Pages 69 and 87.

[297] Catherine Meadows. The nrl protocol analyzer: An overview. *J. Log. Program.*, 26(2):113–131, 1996. Page 69.

[298] A. D. Brucker and S. Mödersheim. Integrating automated and interactive protocol verification. In *Formal Aspects in Security and Trust (FAST)*, volume 5983 of *LNCS*, pages 248–262. Springer, 2009. Pages 69, 88 and 90.

[299] J. Goubault-Larrecq. Finite models for formal security proofs. *Journal of Computer Security*, 18(6):1247–1299, 2010. Pages 69 and 90.

[300] C. Cremer, D. Basin, and S. Meier. Efficient construction of machine-checked symbolic protocol security proofs. *Journal of Computer Security*, 2013. to appear. Page 69.

[301] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology (CRYPTO)*, volume 6841 of *LNCS*, pages 71–90. Springer, 2011. Pages 69 and 72.

[302] Miriam Paiola and Bruno Blanchet. Verification of security protocols with lists: From length one to unbounded length. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust (POST)*, volume 7215 of *LNCS*, pages 69–88. Springer, 2012. Page 69.

[303] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Efficient verification of security protocols using partial-order reductions. *STTT*, pages 173–188, 2003. Pages 70 and 77.

[304] Jonathan Ezekiel and Gerald Lüttgen. Measuring and evaluating parallel state-space exploration algorithms. *Electr. Notes Theor. Comput. Sci.*, 198(1):47–61, 2008. Pages 70 and 89.

[305] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996. Page 70.

[306] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 2008. Page 70.

[307] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 4(2):153–167, 1997. Page 70.

[308] 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. Page 70.

[309] F. Pommereau. *Algebras of coloured Petri nets*. Lambert Academic Publisher, 2010. ISBN 978-3-8433-6113-2. Pages 70 and 89.

[310] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002. Page 71.

[311] R. Boyer and J. Moore. *Metafunctions: proving them correct and using them efficiently as new proof procedures*, chapter The Correctness Problem in Computer Science. NY: Academic Press, 1981. Pages 71 and 72.

[312] Christopher Sprenger. A verified model checker for the modal μ-calculus in coq. In *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 167–183. Springer-Verlag, 1998. Pages 71, 72, 73 and 90.

[313] George C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, pages 106–119. ACM, 1997. Page 71.

[314] Kedar S. Namjoshi. Certifying model checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 2–13. Springer, 2001. Pages 71 and 72.

[315] Doron Peled, Amir Pnueli, and Lenore D. Zuck. From falsification to verification. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2245 of *LNCS*, pages 292–304. Springer, 2001. Pages 71 and 72.

[316] Li Tan and Rance Cleaveland. Evidence-based model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 455–470. Springer, 2002. Pages 71 and 72.

[317] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009. Pages 71 and 72.

[318] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to sat verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *First International Conference, of Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010. Page 71.

[319] John Harrison. Binary Decision Diagrams as a HOL Derived Rule. *Comput. J.*, 38(2):162–170, 1995. Page 72.

[320] Robert R. Schneck and George C. Necula. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In Andrei Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 47–62. Springer, 2002. Page 72.

[321] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In Jifeng He and Masahiko Sato, editors, *Advances in Computing Science - 6th Asian Computing Science Conference (ASIAN)*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000. Pages 72 and 90.

[322] Xavier Rival and Jean Goubault-Larrecq. Experiments with finite tree automata in coq. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2152 of *LNCS*, pages 362–377. Springer, 2001. Pages 72 and 90.

[323] Jonathan Ford and Natarajan Shankar. Formal verification of a combination decision procedure. In Andrei Voronkov, editor, *Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 347–362. Springer, 2002. Pages 72 and 73.

[324] Natarajan Shankar and Marc Vaucher. The mechanical verification of a dpll-based satisfiability solver. *Electr. Notes Theor. Comput. Sci.*, 269:3–17, 2011. Pages 72 and 73.

[325] Natarajan Shankar. Trust and automation in verification tools. In Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *LNCS*, pages 4–17. Springer, 2008. Page 72.

[326] Bruno Barras and Benjamin Werner. Coq in coq. Technical report, INRIA, 1997. Page 72.

[327] Sascha Böhme, Anthony Fox, Thomas Sewell, and Tjark Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In Jean-Pierre Jouannaud and Zhong Shao, editors, *First International Conference of Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 183–198. Springer, 2011. Pages 72 and 73.

[328] Jun Sun, Yang Liu, and Bin Cheng. Model checking a model checker: A code contract combined approach. In Jin Song Dong and Huibiao Zhu, editors, *12th International Conference on Formal Engineering Methods (ICFEM)*, volume 6447 of *LNCS*, pages 518–533. Springer, 2010. Page 72.

[329] J. Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics Masaryk University Brno, 2004. Pages 75 and 89.

[330] 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. Page 78.

[331] Alexandru Iosup, Ozan Sonmez, Shanny Anoep, and Dick Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Symposium on High performance distributed computing (HPDC)*, pages 97–108. ACM, 2008. Page 79.

[332] K. Hinsen. Parallel scripting with Python. *Computing in Science & Engineering*, 9(6), 2007. Page 80.

[333] 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 81, 82 and 83.

[334] Valentin Goranko, Angelo Kyrilov, and Dmitry Shkatov. Tableau tool for testing satisfiability in ltl: Implementation and experimental analysis. *Electr. Notes Theor. Comput. Sci.*, 262:113–125, 2010. Page 82.

[335] 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 82.

[336] 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 83.

[337] Jaco Geldenhuys and Antti Valmari. More efficient on-the-fly ltl verification with tarjan's algorithm. *Theor. Comput. Sci.*, 345(1):60–82, 2005. Page 83.

[338] R. Corin. *Analysis Models for Security Protocols*. PhD thesis, University of Twente, 2006. Pages 84 and 86.

[339] C. Inggs, H. Barringer, A. Nenadic, and N. Zhang. Model checking a security protocol. In *Southern African Telecommunications Network and Applications Conference (SATNAC)*, 2004. Pages 86 and 90.

[340] 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 87.

[341] 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 87 and 88.

[342] 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 87.

[343] N. Dalal, J. Shah, K. Hisaria, and D. Jinwala. A comparative analysis of tools for verification of security protocols. *Int. J. Communications, Network and System Sciences*, 3:779–787, 2010. Page 87.

[344] J. F. Cremers, P. Lafourcade, and P. Nadeau. Comparing state spaces in automatic security protocol analysis. In *Formal to Practical Security*, volume 5458 of *LNCS*, pages 70–94. Springer, 2009. Page 87.

[345] Qurat ul Ain Nizamani and Emilio Tuosto. Heuristic methods for security protocols. In Michele Boreale and Steve Kremer, editors, *Proceedings 7th International Workshop on Security Issues in Concurrency (SECCO)*, volume 7 of *EPTCS*, pages 61–75, 2009. Page 87.

[346] Stylianos Basagiannis, Panagiotis Katsaros, and Andrew Pombortsis. An intruder model with message inspection for model checking security protocols. *Computers & Security*, 29(1):16–34, 2010. Page 87.

[347] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998. Page 88.

[348] Amerson Lin, Mike Bond, and Jolyon Clulow. Modeling partial attacks with alloy. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols Workshop*, volume 5964 of *LNCS*, pages 20–33. Springer, 2007. Page 88.

[349] Reynald Affeldt, Miki Tanaka, and Nicolas Marti. Formal proof of provable security by game-playing in a proof assistant. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security (ProvSec)*, volume 4784 of *LNCS*, pages 151–168. Springer, 2007. Page 88.

[350] Michael Backes and Dominique Unruh. Limits of constructive security proofs. In Josef Pieprzyk, editor, *Theory and Application of Cryptology and Information Security (ASIACRYPT)*, volume 5350 of *LNCS*, pages 290–307. Springer, 2008. Page 88.

[351] Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *J. Autom. Reasoning*, 48(2):159–195, 2012. Page 88.

[352] 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 88.

[353] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998. Page 88.

[354] Yang Liu, Jun Sun, and Jin Song Dong. Scalable multi-core model checking fairness enhanced systems. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering (ICFEM)*, volume 5885 of *LNCS*, pages 426–445. Springer, 2009. Page 88.

[355] A. W. Roscoe and Philippa J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(1):147–190, 1999. Page 88.

[356] Guoqiang LI. *On-the-fly Model Checking of Security Protocols*. PhD thesis, Japan Advanced Institute of Science and Technology, 2008. Page 88.

[357] David A. Basin, Sebastian Mödersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. In Einar Snekkenes and Dieter Gollmann, editors, *8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *LNCS*, pages 253–270. Springer, 2003. Page 88.

[358] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. Tarmo: A framework for parallelized bounded model checking. In Lubos Brim and Jaco van de Pol, editors, *8th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, volume 14 of *EPTCS*, pages 62–76, 2009. Page 88.

[359] Wilfredo Rogelio Marrero. *Brutus: a model checker for security protocols*. PhD thesis, Carnegie Mellon University, 2001. Page 88.

[360] 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 88.

[361] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *Parallel and Distributed Model-Checking (PDMC)*, 2010. Page 88.

[362] Ulrich Stern and David L. Dill. Parallelizing the murj verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. Page 88.

[363] M. Leucker and T. Noll. A distributed model checking tool tailored to Erlang. In *Proc. of Erlang Workshop at PLI'01*, 2001. Page 88.

[364] 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 88.

[365] 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 89.

[366] Ł. Fronc and F. Pommereau. Optimising the compilation of petri net models. In *SUMO'11*, volume 726 of *CEUR*. CEUR-WS, 2011. Pages 89 and 91.

[367] 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. Pages 89 and 97.

[368] Samira Chaou, Gil Utard, and Franck Pommereau. Evaluating a peer-to-peer storage system in presence of malicious peers. In Waleed W. Smari and John P. McIntire, editors, *High Performance Computing and Simulation (HPCS)*, pages 419–426. IEEE, 2011. Page 89.

[369] Bruno Blanchet and Avik Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *Security and Privacy*, pages 417–431. IEEE Computer Society, 2008. Page 89.

[370] Marco Danelutto, G. Di Caprio, and A. Masini. Parallelizing a model checker. In Hamid R. Arabnia, editor, *Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1118–1128. CSREA Press, 1996. Page 89.

[371] C. Pajault. *Model Checking parallèle et réparti de réseaux de Petri colorés de haut-niveau*. PhD thesis, Conservatoire National des Arts et Métiers, 2008. Page 89.

[372] C. Boukala and L. Petrucci. Towards distributed verification of Petri nets properties. In *1st International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS)*, eWiC, pages 15–26. British Computer Society, 2007. Page 89.

[373] Rodrigo Saad, Tacla, Bernard Berthomieu, Silvano Dal Zilio, and François Vernadat. Enumerative Parallel and Distributed State Space Construction. In *ETR09 - École d'été Temps Réel*, 2009. Page 89.

[374] G. Guirado, T. Herault, R. Lassaigne, and S. Peyronnet. Distribution, approximation and probabilistic model checking. In *ENTCS*, volume 135, pages 19–30. Elsevier, 2006. Page 89.

[375] Gianfranco Ciardo, Joshua Gluckman, and David M. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, 1998. Page 89.

[376] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. *STTT*, 11(1):13–25, 2009. Page 89.

[377] Jeremy M. R. Martin and Yvonne Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 1–14, 2000. Page 89.

[378] D. Petcu. Parallel explicit state reachability analysis and state space construction. In *Proceedings of ISPDC*, pages 207–214. IEEE Computer Society, 2003. Page 89.

[379] A. Lluch-Lafuente. implified distributed model checking by localizing cycles. Technical Report 176, Institute of Computer Science at Freiburg University, 2002. Page 89.

[380] 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 89.

[381] 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. Page 89.

[382] 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 90.

[383] 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 90.

[384] 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 90.

[385] Stefan Blom, Bert Lisser, Jaco van de Pol, and Michael Weber. A database approach to distributed state-space generation. *J. Log. Comput.*, 21(1):45–62, 2011. Page 90.

[386] 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 90.

[387] 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 90.

[388] S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In *ENTCS*, volume 89. Elsevier, 2003. Page 90.

[389] 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 90.

[390] Alberto Lluch Lafuente. Simplified distributed ltl model-checking by localizing cycles. Technical Report IFI-2002, Institut fur Informatick, Albert-Ludwigs-Universitat, 2000. Page 90.

[391] J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011. Page 90.

[392] I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In *SPIN workshop*, volume 2648 of *LNCS*, pages 49–74. Springer, 2003. Page 90.

[393] Lubos Brim, Ivana Cerná, and Lukás Hejtmánek. Distributed negative cycle detection algorithms. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, and Wolfgang V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications (PARCO)*, volume 13 of *Advances in Parallel Computing*, pages 297–304. Elsevier, 2003. Page 90.

[394] Lubos Brim, Ivana Cerná, Pavel Moravec, and Jirí Simsa. Accepting predecessors are better than back edges in distributed ltl model-checking. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004. Page 90.

[395] Rodrigo T. Saad, Silvano Dal-Zilio, and Bernard Berthomieu. Mixed shared-distributed hash tables approaches for parallel state space construction. In *Parallel and Distributed Computing (ISPDC)*, pages 9–16. IEEE, 2011. Page 90.

[396] Sami Evangelista, Laure Petrucci, and Samir Youcef. Parallel nested depth-first searches for ltl model checking. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 6996 of *LNCS*, pages 381–396. Springer, 2011. Page 90.

[397] Alfons Laarman and Jaco van de Pol. Variations on multi-core nested depth-first search. In Jiri Barnat and Keijo Heljanko, editors, *Parallel and Distributed Methods in verifiCation (PDMC)*, volume 72 of *EPTCS*, pages 13–28, 2011. Page 90.

[398] Jiri Barnat, Lubos Brim, and Milan Ceska. Divine-cuda - a tool for gpu accelerated ltl model checking. In Lubos Brim and Jaco van de Pol, editors, *Parallel and Distributed Methods in verifiCation (PDMC)*, volume 14 of *EPTCS*, pages 107–111, 2009. Page 90.

[399] Cornelia P. Inggs and Howard Barringer. Ctl* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, 2006. Page 90.

[400] Willem Visser, Howard Barringer, Donal Fellows, Graham Gough, and Alan Williams. Efficient ctl* model checking for analysis of rainbow designs. In Hon Fung Li and David K. Probst, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 105 of *IFIP Conference Proceedings*, pages 128–145. Chapman & Hall, 1997. Page 90.

[401] Mohand Cherif Boukala and Laure Petrucci. Distributed model-checking and counterexample search for ctl logic. *IJCCBS*, 3(1/2):44–59, 2012. Page 90.

[402] Martin Leucker, Rafal Somla, and Michael Weber. Parallel model checking for ltl, ctl*, and lţ. *Electr. Notes Theor. Comput. Sci.*, pages 1–1, 2003. Page 90.

[403] Lubos Brim and Jitka Zidkova. Using assumptions to distribute alternation free mu-calculus model checking. *Electr. Notes Theor. Comput. Sci.*, 89(1):17–32, 2003. Page 90.

[404] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free ţ-calculus. In *SPIN on Model checking of Software*, pages 128–147. Springer-Verlag, 2002. Page 90.

[405] E. Turner, M. Butler, and M. Leuschel. A refinement-based correctness proof of symmetry reduced model checking. In *Abstract State Machines, Alloy, B and Z*, LNCS, pages 231–244. Springer, 2010. Page 90.

[406] J. den Hartog. Towards mechanized correctness proofs for cryptographic algorithms: Axiomatization of a probabilistic hoare style logic. *Sci. Comput. Program.*, 74(1–2):52–63, 2008. Page 91.

[407] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*. Springer, 2009. Page 91.

[408] S. Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Logic and Computation*, 13(6):801–813, 2003. Page 91.

[409] Ming-Hsien Tsai and Bow-Yaw Wang. Formalization of ctl* in calculus of inductive constructions. In Mitsu Okada and Ichiro Satoh, editors, *Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues (ASIAN)*, volume 4435 of *LNCS*, pages 316–330. Springer-Verlag, 2007. Page 91.

[410] Lukasz Fronc and Franck Pommereau. Towards a certified petri net model-checker. In Hongseok Yang, editor, *Programming Languages and Systems (APLAS) - 9th Asian Symposium*, volume 7078 of *LNCS*, pages 322–336. Springer, 2011. Page 91.

[411] Martin Leucker, Thomas Noll, Perdita Stevens, and Michael Weber. Functional programming languages for verification tools: a comparison of standard ml and haskell. *STTT*, 7(2):184–194, 2005. Page 91.

[412] A. Danalis, L. Pollock, and M. Swany. Automatic MPI application transformation with ASPhALT. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2007), in conjunction with IPDPS*, 2007. Page 94.

[413] Chong Li and Gaétan Hains. Sgl: towards a bridging model for heterogeneous hierarchical platforms. *IJHPCN*, 7(2):139–151, 2012. Page 95.

[414] L. G. Valiant. A bridging model for multi-core computing. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag. Page 95.

[415] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, LNCS, pages 102–108. Springer-Verlag, 2000. Page 95.

[416] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010. Page 97.

[417] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? In Lubos Brim and Jaco van de Pol, editors, *Parallel and Distributed Methods in verifiCation (PDMC)*, volume 14 of *EPTCS*, pages 1–17, 2009. Page 97.

[418] 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 97.

[419] Todd R. Andel, G. Back, and Alec Yasinsac. Automating the security analysis process of secure ad hoc routing protocols. *Simulation Modelling Practice and Theory*, 19(9):2032–2049, 2011. Page 97.

[420] *et al* Agarwal, A.K. An experimental study on wireless security protocols over mobile ip networks. In *Vehicular Technology Conference (VTC)*, pages 5271–5275. IEEE Computer Society, 2004. Page 97.

[421] Mathilde Arnaud. *Formal verification of secured routing protocols*. PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, 2011. Page 97.

[422] Véronique Cortier, Jan Degrieck, and Stéphanie Delaune. Analysing routing protocols: Four nodes topologies are sufficient. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust (POST)*, volume 7215 of *LNCS*, pages 30–50. Springer, 2012. Page 97.

[423] Rohit Chadha, Steve Kremer, and Andre Scedrov. Formal analysis of multi-party contract signing. In *Security Foundations (CSFW)*, pages 266–265. IEEE Computer Society, 2004. Page 97.

[424] 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 97.

[425] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. Spins: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002. Page 97.

[426] J. Falcou and J. Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In *International Conference ParCo*, 2007. Page 98.

[427] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Oracle, Java SE 7 edition, 2011. Page 98.

[428] Matthieu Lemerre, Vincent David, and Guy Vidal-Naquet. A communication mechanism for resource isolation. In *Isolation and Integration in Embedded Systems (IIES)*, pages 1–6. ACM, 2009. Page 99.

Softwares

1 **MPI**: `http://www.mpi-forum.org/`

2 **Open-MP**: `http://openmp.org/wp/`

3 **Caraml** project: `http://caraml.free.fr/`

4 **ProPac** project: `wwwpropac.free.fr`

5 **SPREADS** project: `http://spreads.fr/index.html`

6 **BSMLlib**: `http://bsmllib.free.fr`

7 **OCaml**: `http://caml.org`

8 **BSPlib**: `http://www.bsp-worldwide.org/implmnts/oxtool/`

9 **PUB**: `http://www2.cs.uni-paderborn.de/~pub/`

10 **PVM**: `http://www.csm.ornl.gov/pvm/`

11 **OCamlP3L**: `http://ocamlp3l.inria.fr/`

12 **Sketo**: http://sketo.ipl-lab.org/
13 **CGMlib**: http://www.scs.carleton.ca/cgm/
14 **BSPonMPI**: http://bsponmpi.sourceforge.net/
15 **BSP-Core**: http://www.multicorebsp.com/
16 **Hamma**: http://hama.apache.org/
17 **Pregel**: http://googleresearch.blogspot.fr/2009/06/large-scale-graph-computing-at-google.html
18 **NestStep**: http://www.ida.liu.se/~chrke/neststep/index.html
19 **BSPonGPU**: http://kunzhou.net/BSGP/BSGP-package.zip
20 **BSP++**: https://github.com/khamidouche/BSPPP
21 **BSP-Python**: http://dirac.cnrs-orleans.fr/plone/software/scientificpython/
22 **NESL**: http://www.cs.cmu.edu/~scandal/nesl.html
23 **Nepal**: http://homepages.inf.ed.ac.uk/wadler/realworld/nepal.html
24 **Manticore**: http://manticore.cs.uchicago.edu/
25 **Concurrent ML**: http://cml.cs.uchicago.edu/
26 **SAC**: www.sac-home.org/
27 **Data-Parallel Haskell**: http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell
28 **Eden**: http://www.mathematik.uni-marburg.de/~eden/
29 **Gph**: http://www.macs.hw.ac.uk/~dsg/gph/
30 **Hume**: http://www-fp.cs.st-andrews.ac.uk/hume/index.shtml
31 **SISAL**: http://sourceforge.net/projects/sisal/
32 **Erlang**: http://www.erlang.org/
33 **JoCaml**: http://jocaml.inria.fr/
34 **Alice ML**: http://www.ps.uni-saarland.de/alice/
35 **Scala**: http://www.scala-lang.org/
36 **Oz-Mozart**: http://www.mozart-oz.org/home/doc/index.html
37 **Split-C**: http://www.eecs.berkeley.edu/Research/Projects/CS/parallel/castle/split-c/
38 **UPC**: http://upc.gwu.edu/
39 **Cilk**: http://supertech.csail.mit.edu/cilk/
40 **Fortress**: http://projectfortress.java.net/
41 **ZPL**: http://www.cs.washington.edu/research/zpl/home/index.html
42 **Orca**: http://www.cs.vu.nl/orca/
43 **Co-Array Fortan**: http://www.co-array.org/
44 **Occam**: pop-users.org/wiki/occam-pi
45 **Charm++**: http://charm.cs.illinois.edu/software
46 **Plasma**: http://plasma.camlcity.org/plasma/index.html
47 **Tom**: http://tom.loria.fr/wiki/index.php5/Main_Page
48 **MGS**: http://mgs.spatial-computing.org/index.html
49 **OCamlFloat**: http://www.mirrorsky.com/ocaml/
50 **psilab**: http://psilab.sourceforge.net/
51 **lacaml**: http://hg.ocaml.info/release/lacaml
52 **scipy**: http://www.scipy.org/
53 **TLA+**: http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html
54 **MPI-SPIN**: http://vsl.cis.udel.edu/mpi-spin/index.html
55 **ISP**: http://www.cs.utah.edu/formal_verification/ISP-Release/
56 **DAMPI**: http://www.cs.utah.edu/formal_verification/DAMPI/
57 **why** : http://why.lri.fr/index.html
58 **Coq**: http://coq.inria.fr/
59 **PVS**: http://pvs.csl.sri.com/
60 **Isabelle**: http://isabelle.in.tum.de/
61 **HOL**: http://hol.sourceforge.net/
62 **Simplify**: http://www.hpl.hp.com/downloads/crl/jtk/index.html
63 **Alt-Ergo**: http://ergo.lri.fr/
64 **Z3**: http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html
65 **CVC3**: http://cs.nyu.edu/acsys/cvc3/
66 **Yices**: http://yices.csl.sri.com/
67 **Vampire**: http://www.vprover.org/
68 **Who**: https://github.com/kanigsson/who

69 **Boogie**: http://research.microsoft.com/en-us/projects/boogie/
70 **VCC**: http://research.microsoft.com/en-us/projects/vcc/default.aspx
71 **HPCBugDatabase**: http://www.hpcbugbase.org/
72 **TASS**: http://vsl.cis.udel.edu/tass/index.html
73 **FEVS**: http://vsl.cis.udel.edu/fevs/index.html
74 **Heap-hop**: http://www.lsv.ens-cachan.fr/Software/heap-hop/
75 **AVISPA**: http://www.avispa-project.org/
76 **Scyther**: http://people.inf.ethz.ch/cremersc/scyther/
77 **Tamarin**: http://www.infsec.ethz.ch/research/software/tamarin
78 **ProVerif**: http://www.proverif.ens.fr/
79 **NRL**: http://maude.cs.uiuc.edu/tools/Maude-NPA/
80 **EAL7**: http://www.commoncriteriaportal.org/
81 **PAT**: http://www.comp.nus.edu.sg/~pat/
82 **SPORE**: http://www.lsv.ens-cachan.fr/Software/spore
83 **CAPSL**: http://www.csl.sri.com/users/millen/capsl/
84 **Murphi**: http://www.cs.utah.edu/formal_verification/Murphi/
85 **μCRL**: http://homepages.cwi.nl/~mcrl/
86 **Eddy**: http://www.cs.utah.edu/formal_verification/EddyMurphi/
87 **Spin**: http://spinroot.com/spin/whatispin.html
88 **Frama-C**: http://frama-c.com/
89 **JRockit**: http://docs.oracle.com/cd/E23009_01/index.htm

Glossary

ACI: *Action Concertée Incitative.*
ACI (now call ANR) is a past french organisation for research funding.

ANR: *Agence National pour la Recherche.*
ANR (previously called ACI) is one the french organisations for research funding.

BSML: *Bulk-Synchronous Parallel ML.*
A library for programming BSP algorithm in a functionnal manner.

BSP: *Bulk Synchronous Parallelism.*
A structured and coarsed-grained model of parallel computations.

BSPlib: *Bulk Synchronous Parallel programming library.*
The first C library for BSP computing.

CFA: *Control Flow Analysis.*
CFA is a static code analysis technique for determining the control flow of a program.

CPS: *Continuation-passing style.*
CPS is a style of programming in which control is passed explicitly in the form of a continuation.

CTL: *Computation tree logic.*
CTL is a temporal and branching-time logic.

DFS: *Depp First Search.*
An algorithm for traversing a structure by starting at the root and exploring as far as possible along each childs before backtracking.

FFT: *Fast Fourier Transform.*
FFT is an efficient method to compute a specific kind of discrete transform, used in Fourier analysis.

GPH: *Glasgow Parallel Haskell.*
GPH is a parallel extension of the non-strict functional language Haskell with thread-based semi-explicit parallelism.

GPU: *Graphics Processing Unit.*
GPU are specialized electronic circuits designed to accelerate the building of images. They are efficient when processing of large blocks of data is done in parallel.

HPC: *High-Performance Computing.*
The fact of using a massive number of processors, a supercomputer.

LTL: *Linear Temporal logic.*
LTL is a modal temporal logic with modalities referring to time.

LTS: *Labelled Transition System.*
A direct-graph (with labels on vertices) that can represent the execution of a system.

MPI: *Message Passing Interface.*
MPI is a standardized and portable message-passing system to function on a wide variety of parallel computers.

OCAML: *Objective Caml.*
OCAML is a general-purpose programming strongly typed language which supports functional, imperative, and object-oriented programming styles.

PCC: *Proof-Carrying Code.*
A family of techniques to generate certificates (in a theorem prover) of properties (invariant) of given transformations (compiler) or analysis.

P3L: *Pisa Parallel Programming Language.*
P3L is a data-flow skeleton based coordination language.

PDE: *Partial Differential Equation.*
PDE is a differential equation that contains unknown multivariable functions and their partial derivatives.

ProPac: *PROgrammation PAralléle Certifée.*
ANR project for developping tools and languages for certified BSP computations.

PUB: *Paderborn University BSPlib.*
An extension of the BSPlib for mainly high-performance communications and subgroup synchronisations.

PVM: *Parallel Virtual Machine.*
PVM is a library for parallel networking of heterogeneous computers.

SCC: *Strongly Connected Component.*
The SCC of a directed graph are its maximal strongly connected subgraphs — a graph is strongly connected if there is a path from each vertex in the graph to every other vertex.

SHM: *SHared Memory.*
SHM is the Unix implementation of traditional shared memory concept.

SMT: *Satisfiability Modulo Theories.*
SMT is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.

SPREADS: *Safe P2p REaliable Architecture for Data Storage*
ANR project to study and design a highly dynamic secure P2P storage systems on large scale networks like the internet.

SPMD: *Single Program Multiple Data.*
SPMD is technique employed to achieve parallelism; Tasks are split up and run simultaneously on multiple processors.

TDS: *Tridiagonal System Solver.*
Solve the tridiagonal system $A.x = b$ where A is a $n \times n$ tridiagonal matrix representing coefficients, x a vector of unknowns and b a right-hand-side vector.

VCG: *Verification Condition Generator.*
A tool for deductive verification of programs; it takes an logical annotated program and generates conditions (to be proved) that ensurring the correctness of the annotated program.

Curriculum Vitae

Organization of research

Project

- 2007–2010: the SPREADS project (ANR Securité 2008)
- 2007–2010: the VEHICULAIRE project (Fondation Digiteo, Region Paris)
- 2004–2007: the ProPac project (ACI Young researchers 2004)
- 2002–2004: the CARAML project (ACI GRID 2001)

Reviews

- Journals:
  - SCPE ("Scalable Computing: Practice and Experience"), SWPS Publisher (previously named PDCP, "Parallel and Distributed Computing Practices", NOVA Sciences Publisher);
  - CLSS ("Computer Languages, Systems and Structures"), Elsevier Publisher;
  - HPCN ("High Performance Computing and Networking"), InderScience Publishers;
  - TPDS ("Transactions on Parallel and Distributed Systems"), IEEE Publishers;
  - TSI ("Technique et Science Informatiques"), Hermes Lavoisier Publishers;
  - PPL ("Parallel Processing Letters"), World Scientific Publisher.
- Conferences:
  - ICCS ("International Conference on Computational Science"), 2007-2010;
  - HPCS ("High Performance Computing and Simulation"), 2011 and 2012;
  - PDCN ("Parallel and Distributed Computing and Networks"), 2011.
  - Symposium TFP ("Trends in Functional Programming") 2005;
- Workshops
  - PaPP ("aPplications of declArative and object-oriented Parallel Programming") 2004–2006 affiliated to ICCS;
  - APDCM ("Advances in Parallel and Distributed Computational Models") 2009 affiliated to IPDPS ("Parallel and Distributed Processing Symposium");

PC member

- Conference ICCS 2011-13
- Workshop HLPP ("High-level Parallel Programming and Applications") 2013;
- Workshops PaPP 2007, 2008, 2011 and 2012;
- Workshops WLPP ("Language-Based Parallel Programming") 2007, 2009, 2011 and 2013; affiliated to the conference PPAM ("Parallel Processing and Applied Mathematics");
- Workshop HPDFA ("High-Performance and Distributed Computing for Financial Applications") 2010.

Organizer

- With Dr. Anne Benoit (ENS Lyon) of the workshops PAPP 2009 and 2010 (part of ICCS);
- With Pr. Gaétan Hains (University of Paris-East) and Pr. Kevin Hammond (University of St. Andrews) of the workshop HLPGPU ("High-level programming for heterogeneous and hierarchical parallel systems") 2011 affiliated to the conference HiPEAC ("High-Performance and Embedded Architectures and Compilers").

Guest Editor

Co-editor with Pr. Gaétan Hains and Pr. Kevin Hammond of the journal PPL (Volume 22, Number 2, June 2012) for a selection of papers of the workshop HLPGPU 2011.

Research supervision

Master Internship

Arthur Hidalgo: "Verification of security protocols: machine-checked proof of the algorithms", master recherche NSI, Orsay, 2012;

Jean Fortin: "Optimisation Certifiée de code BSP", master recherche MPRI (from ENS Lyon), 2008;

Ilias Garnier: "Nouvelle implantation de la primitive BSML de superposition et application à l'implantation de squelettes algorithmiques", master recherche MPRI Paris VII, 2008;

Otmane Bouziani: "Implantation d'une bibliothèque de programmation fonctionnelle BSP dans un environnement de méta-computing", master recherche SSI, Paris 12, 2006;

David Billiet: "BSML: Implémentation modulaire et prévision de performances", master recherche NSI, 2004, Orsay, co-director with Pr. Frédédric Loulergue;

Doctoral Thesis

Jean Fortin: "Mechanized Deductive Verification of BSP programs: Semantics and Cases Studies", defence in march 2013;

Michael Guedj: "BSP Algorithms for LTL & CTL* Model Checking of Security Protocols" co-director with Pr. Franck Pommereau (IBISC Evry); defence in October 2012 with the following jury:

- Pr. Catalin Dima, president, University of Paris-East
- Pr. Frédéic Loulergue, reviewer, University of Orléans
- Pr. Jean-François Pradat-Peyre, reviewer, University of Paris VI
- Pr. Laure Petrucci, University of Paris-North
- Pr. Hanna Klaudel, University of Evry
- Pr. Gaétan Hains, University of Paris-East
- Pr. Franck Pommereau, co-director, University of Evry
- Dr. Frédéric Gava, co-director, University of Paris-East

Louis Gesbert: "Développement systématique et sûreté d'exécution en programmation parallèle structurée"; co-director with Pr. Frédéric Loulergue (LIFO Orléans); defence in March 2009 with the following jury:

- Pr. Olivier Michel, president, University of Paris-East
- Pr. Emmanuel Chailloux, reviewer, University of Paris VII
- Pr. Jocelyn Sérot, reviewer, University of Clermond-Ferant
- Pr. Zhenjiang Hu, National Institute of Informatics, Tokyo
- Pr. Frédéric Loulergue, director, University of Orléans
- Dr. Frédéric Gava, co-director, University of Paris-East

Résumé. Certains problèmes nécessitent des performances que seules les machines massivement parallèles et distribuées peuvent offrir. Néanmoins, programmer de telles architectures est une tâche difficile. Nous trouvons plusieurs raisons comme la complexité intrinsèque de ces architectures (notamment les inter-blocages et les accès concurrents), un déficit de formation ou le manque d'outils pour le test et la preuve des programmes parallèles. Mais surtout, les structures de haut-niveau comme les patrons algorithmiques sont trop souvent absentes car méconnues. Pourtant elles facilitent l'écriture des programmes tout en leur assurant une plus grande sûreté d'exécution et potentiellement leurs corrections vis-à-vis de leurs spécifications. De plus, l'accès à ces machines se fait via des réseaux non-sûrs. Garantir la sécurité des échanges entre les utilisateurs et les centres de calculs est aussi nécessaire que de pouvoir vérifier la correction des programmes.

Dans ce travail, nous nous intéressons à un modèle de calcul parallèle, appelé BSP, qui permet de structurer l'exécution des programmes. Le manuscrit est organisé en trois parties, chacune correspondant à une thèse encadrée par l'auteur, dans le cadre de projets de recherche.

Dans un premier temps, nous étudierons un langage fonctionnel pour la programmation BSP, appelé BSML, notamment ses primitives, son implantation, le typage des programmes BSML qui garantit la sûreté d'exécution, une méthodologie pour la preuve de programmes dans l'assistant de preuves COQ, ainsi que diverses extensions, comme la gestion d'exceptions distribuées. Plusieurs applications sont données, notamment, l'implantation de divers patrons algorithmiques.

Dans un second temps, nous décrivons le développement d'un outil pour la preuve déductive d'algorithmes BSP, c'est-à-dire annoter le code par des assertions logiques, puis générer des obligations de preuve qui valident ou non les assertions. Cet outil est une extension d'un outil de vérification déductive déjà existant, appelé WHY. BSP-WHY fonctionne par une transformation du code parallèle BSP (et des assertions logiques) en du code WHY purement séquentiel. Des sémantiques opérationnelles effectuées à l'aide de l'assistant de preuves COQ sont aussi décrites. Des exemples sont aussi donnés afin tester la fiabilité de l'approche.

Dans un dernier temps, nous exploitons la nature bien structurée de protocoles de sécurité et nous la faisons correspondre au modèle BSP afin d'obtenir des algorithmes parallèles efficaces de model-checking des protocoles de sécurité. Nous considérons le problème de la vérification de formules logiques (temporelles) LTL et CTL* sur des protocoles de de sécurité. Les algorithmes BSP de calculs de l'espace des états des protocoles seront aussi vérifiés formellement avec l'outil BSP-WHY décrit dans la précédente partie du manuscrit.

Mots clefs. Parallélisme, Programmation fonctionnelle, Vérification formelle, Protocoles de sécurité.

Abstract. Some problems require performance that only massively parallel and distributed machines can offer. However, programming such architectures is a difficult task. Several reasons can be highlighted for that, such as the intrinsic complexity of these architectures (possibility of deadlocks and data-races), a lack of training of the programmers and a lack of tools to test and to prove the correctness of parallel programs. But, first and foremost, high-level structures such as algorithmic skeletons (patterns of parallel computations) are too often unused, because they are not known by most programmers. However, they facilitate the writing of parallel programs while ensuring them a greater safety and, potentially, their correctness. In addition, access to these supercomputers is performed using unsecured networks. Ensure the security of exchanges between users and data-centers is as important as being able to verify the correctness of programs.

In this work, we investigate a model of parallel computations, called BSP, which allows a structured execution of programs. The manuscript is organized in three parts, each corresponding to a doctoral thesis supervised by the author, within the framework of research projects.

In a first part, we study a functional language for BSP programming, called BSML, including its primitives, its implementation, the type system of BSML programs that guarantees their safety of execution, a methodology for proving programs in the COQ theorem prover and various extensions such as parallel exception handling. Several applications are given, including the implementation of various algorithmic skeletons.

In a second part, we describe the development of a tool for deductive verification of BSP algorithms, that is to say, to annotate the code with logical assertions and to generate proof obligations that validate or not the aforesaid assertions. This tool is an extension of an already existing deductive verification tool, called WHY. BSP-WHY consists in transforming the BSP parallel code (and logical assertions) into purely sequential WHY one. Operational semantics, developed using the COQ theorem prover, are also described. Examples are also given to test the reliability of the approach.

In the last part, we exploit the structured nature of many security protocols and match it to the BSP model, in order to obtain efficient parallel algorithms to model-check security protocols. We consider the problem of the verification of logical (temporal) formulas LTL and CTL* on security protocols. State-space BSP algorithms will also be formally checked using the BSP-WHY tool described in the previous part of the manuscript.

Keywords. Parallelism, Functional Programming, Formal verification, Security protocols.