Towards Verified Cloud Computing Environments

Frédéric Loulergue*, Frédéric Gava†, Nikolai Kosmatov‡ and Matthieu Lemerre§
* LIFO, University of Orléans, Orléans, France
Frederic.Loulergue@univ-orleans.fr
† LACL, University of Paris East Créteil, France
Frederic.Gava@univ-paris-est.fr
‡ CEA, LIST, Software Safety Laboratory, 91191 Gif-sur-Yvette, France
Nikolai.Kosmatov@cea.fr
§ CEA, LIST, Embedded Real-Time System Lab, 91191 Gif-sur-Yvette, France
Matthieu.Lemerre@cea.fr

***Abstract*—As the usage of the cloud becomes pervasive in our lives, it is needed to ensure the reliability, safety and security of cloud environments. In this paper we study a usual software stack of a cloud environment from the perspective of formal verification. This software stack ranges from applications to the hypervisor. We argue that most of the layers could be practically formally verified, even if the work to verify all levels is huge.**

***Keywords*—Formal verification, Cloud computing, Hypervisor**

I. INTRODUCTION

With the development of mobile and internet applications, cloud computing becomes more and more important. More and more of our data is in the cloud. It is thus necessary to have reliable, safe and secure cloud environments.

If the certification of programs in critical systems is an old concern, there is a recent trend in this area to *formally verify* both the programs and the tools used to produce them [1] (and even the tools used to analyse them) using automated and interactive provers, i.e. to have a program and a machine-checked proof that this program meets its specifications. Verified programs could take several forms.

One could use a methodology based on the specification of software and refinement down to an implementation, such as the B method [2]. One could also use methods based on Hoare logic and associated tools. For example in [3] the specifications are pre- and post-conditions of the program, and the programmer should add some invariants to help the verification condition generator and the automated provers to prove the correctness of the program.

Another possibility is to take advantage of the fact that the logic of some interactive provers (such as Coq [4], [5]) contains a functional programming language. One writes the program to verify as a functional program, and proves that the function meets its specification (usually written in the logic of the proof assistant that corresponds to usual mathematical logic). A use case of such an approach is [6] where the program is a compiler, i.e. a function from source code abstract syntax trees to assembly code, that is proved to preserve the semantics of the source program. An actual compiler is then extracted [7] from the Coq development as an OCaml [8], [9] program and compiled to native code.

This work is partly supported by the INEX project funded by the *Conseil Général du Loiret*.

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 could go wrong. A verified compiler itself makes some assumptions 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.

In the area of distributed systems (excluding work on distributed algorithms), the focus is often on security rather than functional correctness. Security is of course important, but security properties can rely on some other properties at different levels of the environment. From this point of view verifying software with respect to functional specifications is a sub-domain of verifying security properties. But verifying the correctness of a program with respect to a functional specification is interesting by itself and is not necessarily related to security matters. Thus we mostly consider in this paper formal verification of software with respect to functional specifications.

A usual software stack in cloud computing environments is depicted in figure 1. In this paper we argue that verification at all these levels is possible and existing work makes us think it is manageable, even if it is not at all in the reach of a single team effort. We discuss each level of the software stack as follows.

At the application level, we will first consider only structured applications, such as MapReduce [10] and Pregel [11] applications, in their Hadoop [12] implementations (respectively MapReduce and Hama). However we will consider two ways of ensuring the correctness of programs: constructive methods, where a program is derived from a specification,

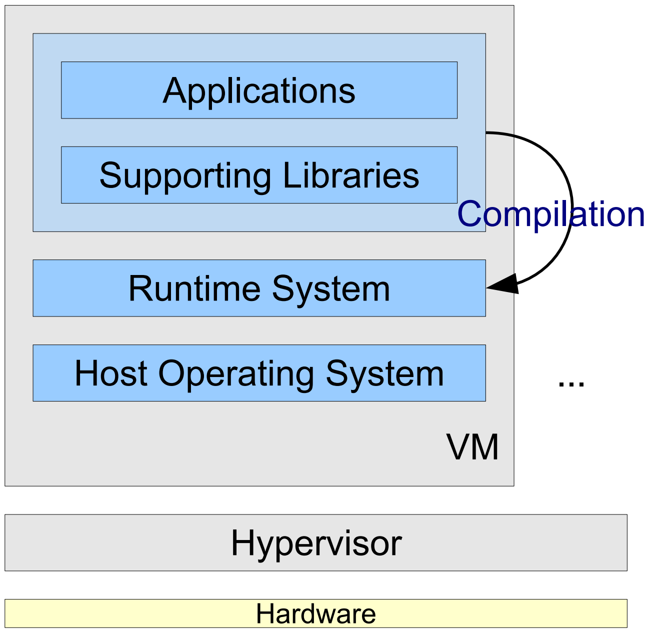

Figure 1. Software Stack

i.e. is correct by construction; a posteriori methods where a program is first written, then proved correct with respect to a specification (section II).

For executing the applications, first we need to compile them: we will consider here only compilation towards byte-code for the JVM [13] as Java is popular for Cloud computing platforms in particular MapReduce. We will nevertheless consider two languages: Java and Scala [14]. Second we need a run-time system: this system consists of a JVM but also of compiled versions of the supporting libraries such as MapReduce (section III).

The JVM runs inside an operating system, itself embedded in a virtual machine running on top of an hypervisor. We will consider here a special case of JVM, such as Oracle's JRockit Virtual Edition [15], 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 (section IV).

We conclude and give the first directions of research in section V.

II. VERIFIED APPLICATIONS

We discuss the verification of programs that rely on two programming libraries for cloud computing: the well-known MapReduce, and Pregel/Hama. Two kinds of verification methods are considered: constructive methods (section II-A) or a posteriori methods (section II-B).

MapReduce [10] was initially developed for simple computations such as page-ranking but for a large set. It is inspired by the algorithmic skeleton paradigm [16]–[18]. In a MapReduce application, the computation proceeds in two phases: the map phase and the reduce phase. The programmer of the application has to specify two functions to be given to the framework: the map function and the reduce function. The former takes key-value pairs as input and outputs key-value pairs of a possibly different type. The map function is applied in parallel to chunks of the input data. The output key-value pairs produced by the application of the map function are sorted by key and all values with the same key are grouped together. These new key-values pairs are given as input to the reduce function that usually reduces, for each key, the set of values to only one value. This is also done on chunks of the reduce phase input data.

Google's Pregel [11] is a new paradigm for large-graph processing. It is graph oriented: each vertex performs a computation where it can send data to other vertices and take into account data received from other vertices. This kind of computing is done until every vertex votes the end of the execution. Each operation on vertices should have a purely functional specification since there is no order of execution. Pregel has thus a BSP [19] like runtime execution. Each stage of computation of Pregel corresponds to a BSP super-step. Page-ranking and shortest paths are natural applications for Pregel and more complicated algorithms can be developed as small-world graph partition (for social networks) or bi-graph decompositions, *etc.*

Hama [20] is a Apache project language inspired from Pregel and BSP computing for Java which mainly provides the classical BSP's send/received operations with explicit barriers of synchronisation. Only the low-level details of execution which is using what is called a BSP Master for scheduling the computations over multi-core architectures is really a change with respect to classical BSP libraries. It handles a large variety of computations (all BSP algorithms that have been developed over many years, see [21] for an overview) but it is currently mainly used for graph partitioning for social networks. Communications are performed with side-effects. Thus a posteriori verification methods seem more suitable than constructive methods.

A. *Constructive Methods*

Transformational programming [22], [23] is a methodology that offers some scope for making the construction of efficient programs more mathematical. Program calculation is a kind of program transformation based on the theory of Constructive Algorithms [24]–[26]. An efficient program is derived step-by-step through a sequence of transformations that preserve the meaning and hence the correctness. With suitable data-structures, program calculation can be used for writing parallel programs [27]–[29].

Constructive algorithms are often related to functional programming [30]. Higher-order combiners are used to write a specification, in a way that is not efficient. This specification, that is a functional program, is then refined into a more efficient functional program. An example of such a derivation could be found in [31]. In the case of parallel programming, the final result is usually still a sequential functional program (in Haskell notation [32], [33]). There is a final step that translates this functional program into a parallel program with algorithmic skeletons (usually written in C++, for example [34], [35]): this translation is not verified. One exception is [36], [37] where the derivation process is done within the Coq proof assistant [5], and a Bulk Synchronous Parallel ML [38], [39] program is *extracted* from the Coq development [7].

In the case of MapReduce programs, a semantics analysis of Google's MapReduce has been conducted in [40]. [41] extends this analysis and uses it as a basis for a constructive method for building Java Hadoop MapReduce programs. The constructive method is further extended in [42]. In this work if the formal basis is sound, there is no verification that the arguments of the derivation framework, written in Java, actually verify the conditions imposed by the formal framework. Moreover there is actually no guarantee that the MapReduce implementation actually fulfills the functional specification of [40] or [41].

To have a more reliable way to construct MapReduce programs, it would be better to derive programs within a proof assistant, then to extract programs from the proof assistant development. There is currently no proof assistant that can produce Java programs. However the Scala programming language could be considered as a functional language. Thus it is appropriate as a target language for proof assistants (Isabelle [43] could produce Scala programs, and in Feburary 2012, a team released the first version of a partial extraction module for the Coq proof assistant [44]). Moreover it compiles to the JVM thus allows to directly use Java MapReduce and Pregel/Hama implementations.

Thus the semantics of MapReduce should be axiomatised in a functional way, the derivation done using this semantics, and extracted programs would depend on the Scala layer on top of MapReduce. Of course, we should verify that the implementation of this Scala layer, as well as the Java implementations of the structured parallel programming libraries, are indeed satisfying the axiomatisation used using the derivation process. We discuss this issue in section III-A.

Axiomatising the semantics of Pregel is possible and would allow us to extract certified graph algorithms. A step by step derivation of graph algorithms from the specification could be done using variants of the BH skeleton [37] (which manipulates distributed lists) since graphs can be considered in a logical point of view as lists of lists of vertices.

B. *A Posteriori Verification*

Hoare logic [45] which is a discipline for *annotating* programs with logical formulae (called assertions) and for extracting logical formulae (called proof obligations) from such programs, can be used for reasoning rigorously about the correctness of programs.

For sequential languages, the traditional way for verifying programs is based on a Verification Condition Generator (VCG). The user annotates the source code with assertions that describe pre- and post-conditions as well as invariants. The VCG tool extracts proof obligations from programs with annotations. The program is correct, i.e. it is guaranteed to satisfy its specification, if the proof obligations can be proved correct. The Boogie [46] and Why [47] systems follow this approach. Why generates proof obligations from an annotated program. These proof obligations can then be sent to automated provers, such as Alt-Ergo [48], and if they fail, to an interactive theorem prover such as Coq. It is to notice that recent work [49] has even studied the verification of the verification condition generators.

In the case of MapReduce, the approach of [50] is similar to the approach of [51], [52] in the context of functional parallel programming. They take advantage of the fact that the Coq proof assistant logic contains a functional language. To allow the writing of MapReduce programs in Coq and reasoning on them, the authors axiomatise the MapReduce functions. The Coq programs are then extracted to Haskell programs that are themselves linked to Hadoop MapReduce through Hadoop Streaming. However the efficiency of the obtained programs is not evaluated. In the same paper they use the Krakatoa/Why [47] framework to reason about Java MapReduce programs.

We think that the approach is interesting and should be mixed with [41] to obtain a framework where one could derive and/or write MapReduce programs within Coq and extract them. However, it would be best to extract Java programs, or at least programs that could be linked to Hadoop implementation without too much overhead such as Scala; or to extract programs in languages that have an Hadoop-like implementation, such as OCaml with the Plasma [53] framework. In the former case, Scala is an interesting choice.

III. VERIFIED COMPILATION AND EXECUTION

A. *Verified Supporting Libraries*

In a MapReduce or Pregel application, the supporting libraries are a large part of the application semantics. Therefore, these libraries, as Java programs, should be proved correct with respect to functional specifications (or axiomatisations used for reasoning about MapReduce and Pregel programs) such as [40].

However, contrary to the case of MapReduce arguments – where the Map and Reduce programs are sequential Java programs – the implementations of the MapReduce and Pregel/Hama libraries are both concurrent and distributed.

To handle the concurrent aspects in the verification, logics that deal with threads and synchronisation primitives are necessary [54]. Of course the implementations of such libraries rely often on parts of the Java library API implemented in C and linked with the Java Native Interface (JNI). The JNI calls refer to low level implementations that in turn calls low level services of the underlying extended JVM or (a library based on calls to) the hypervisor. At this level, the semantics of such low-level functions and OS calls would be axiomatised: only their functional specifications should be used. However during the verification of the JVM (section III-C) and of the hypervisor (section IV), one would check that the implementations indeed follows these specifications.

Distributed aspects are also very important. Indeed the Hadoop framework relies on several software components in particular the Hadoop Distributed File System (HDFS), the job-tracker which coordinates the jobs run and the task-trackers which handle the execution of the (map and reduce) tasks that have been created to perform the job. HDFS and the job-tracker both deal with replication to ensure fault tolerance and high-performance. The parametrisation of the framework is also very high.

A strategy for proving the correctness of such a framework should be progressive: in a first step a functional specification of these components should be designed, with few parameters. Then in a second step, one needs to prove that each component implementation indeed satisfies the functional specification. A first version of the proof may be done with assuming that there is no fault during execution. A second version of the proof should state clearly the hypothesis about the faults. Later on, additional parameters could be added to the verified framework.

B. *Verified Compilation*

Although the interest in the verification of compilers is not new, it is only recently that fully formally verified compilers for large subsets of widely used programming languages were developed, in particular the CompCert compiler [6], [55]–[57] for the C language compiled to ARM, PowerPC or x86 assembler. Such a verified C compiler is needed for the compilation of the hypervisor and for the implementation of the JVM. However for the upper layers of the considered software stack, a formally verified Java compiler is needed. A verified compiler for a small subset of Java is presented in [58]. To handle not only user arguments to the MapReduce and Hama frameworks, but also the implementation of the framework, this subset is not sufficient.

The Java language and the byte-code of the JVM are not very different, and the usual Java compilers do not perform complicated optimisations. The compilation of the sequential part of the Java language to JVM byte-code may not be one of the very difficult tasks. However the Java language specification [59] includes threads: the formalisation of the full specification will be more complicated that the sequential subset and in particular should deal with Java memory model. On the other hand, the Java Virtual Machine specification [13] does not have any instruction for manipulating threads. A look at the implementation of the Thread class of Java API reveals that this implementation relies on native code through JNI. One could imagine that such calls could be given a functional specification for formalising Java's semantics and that the native code would be proved correct with respect with this specification. This is conceivable for some native code API but not for threads. Indeed in this case the native code interacts with the JVM in a non trivial way: it changes the internal state of the JVM. Therefore the formalisation of a multi-threaded JVM should contained both a formalisation of the JVM specification but also a full formalisation of the Thread class (and related classes), including native code. This is the only way in order to be able to prove the correctness of the compilation of multi-threaded Java programs.

The Scala language is also compiled to JVM byte-code. In [60], the author studies the correctness of the compiler passes, but to our knowledge it is not formalised in proof assistant. If Scala programs are to be derived from specifications, a verified Scala compiler would be necessary to a full verified chain.

It is to be noticed that this line of work is not specific to cloud environments: any system using Java or Scala could benefit from certified versions of the compilers.

C. *Verified Runtime System*

The Java and Scala compilers produce Java byte-code to be executed by a Java Virtual Machine. In case we want to avoid a host operating system in a virtualised environment, this JVM could be extended so it could be run directly by the hypervisor.

Existing JVM implementations are usually written in multi-threaded C or C++. Thus proving the correctness of an extended version of one of these implementations would be a huge work. Most of JVM instructions may be not very difficult to prove correct in a sequential and non-optimised version of a JVM, but still a JVM has other components such as the garbage collector, that is not easy to prove correct (some simple GCs have been proved correct, for example [61]). In the case of parallel JVM, the garbage collector will be concurrent and thus even more difficult to prove correct.

Moreover, to provide efficient executions, the JVMs rely on Just-In-Time compilation. There is preliminary work on the verification of JIT [62], but there is still a lot of work to do to attain for example a coverage of real JVMs similar to the coverage provided by the verified CompCert compiler with respect to usual optimising C compilers. In the case of the CompCert compiler, the performance of the programs compiled with CompCert are close to the performances of programs compiled with `gcc` at the first and second level of optimisation, and better in some cases. Verified software could be almost as efficient as non verified software.

The remark about the compilers applies also to the JVM: a verified JVM is interesting for cloud computing but also in other contexts. Only the extensions needed to run the JVM directly on top of the hypervisor would be specific to cloud environments.

IV. VERIFIED HYPERVISOR

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 functionalities.

Anaxagoros [63] is a secure microkernel that is also capable of virtualising preexisting operating systems, for example Linux virtual machines. It is capable of executing hard real-time tasks or operating systems, for instance the PharOS real-time system [64], securely with non real-time tasks, on a single chip.

This goal has required to put a strong emphasis on security in the design of the system, and not only on traditional "behavioural" security (isolation and access control to protect confidentiality and integrity) but also on availability (being able to slow down or steal resources from another task is considered a breach in security).

As it is a microkernel, Anaxagoros is the only piece of code that requires to run in the privileged mode of the CPU in an Anaxagoros-based system. Every piece of code that can be moved out of the kernel is placed in a separated user-level *service*, with limited rights.

This approach contributes to TCB minimisation in two ways:

- first, the kernel, which is the only globally trusted piece of code, is minimized,
- second, as services are isolated, their faults does not affect applications that do not require them. For instance, a bug in a network stack would not affect a task that does not use the network.

For safety and concurrency reasons [63], the interface of the kernel and the main user services is low-level, close to the hardware (this is contrary to other microkernel approaches which attempt to provide a generic interface that abstracts the hardware). This approach also allows to classify Anaxagoros as an exokernel [65] or as an hypervisor.

This approach also contributes to TCB minimisation: as the interface provides no abstraction, the code of the kernel and services becomes much simpler, as it only has to check that the required hardware operations are permitted.

The kernel generally strictly enforces Saltzer and Schroeder behavioural security principles [66]. In addition to minimising TCB, the kernel provides protection domains using the machine's virtual memory mechanisms and controls access to shared services using capabilities.

The kernel and services are designed to prevent availability attacks, which are a problem often ignored in conventional system design. In particular the *denial of resources* attack can be made when a task can issue requests that make the kernel or a service allocate a resource (e.g. memory): by issuing a sufficient number of requests, the system can run out of memory. New resource security mechanisms and principles have been built in Anaxagoros to avoid this kind of attack (for instance the kernel does not allocate any memory, while still allowing dynamic creation of new virtual machines).

A recent work [67] presented rigorous, formal verification for the OS microkernel seL4, allowing devices running seL4 to achieve the EAL7 evaluation level of the Common Criteria [68]. Another formal verification of a microkernel was reported in [69]. In both cases, the verification used interactive, machine-assisted and machine-checked proof with the theorem prover Isabelle/HOL. Although interactive theorem proving requires human intervention to construct and guide the proof, it has the benefit to serve a general range of properties and is not limited to specific properties treatable by more automated methods of verification such as static analysis or model checking.

The formal verification of a simple hypervisor [70] uses VCC [71], an automatic first-order logic based verifier for C. The underlying system architecture is precisely modeled and represented in VCC, where the mixed-language system software is then proved correct. Unlike [67] and [69], this technique is based on automated methods.

The purpose of our ongoing work is the formal verification of the Anaxagoros hypervisor. Our approach is based on the specification of the code in the ACSL [72] specification language and proving it using the plugins Jessie and WP for program proving of the Frama-C tool [73]. We are currently working on the proof of the virtual memory management module, one of the most critical modules.

An important future work direction is the verification of a concurrent hypervisor. For instance, the verification in [67], [70] was carried out for a sequential version. This research direction is extremely important for an OS or a hypervisor since concurrency naturally appears both for parallel execution on a multi-core architecture and for non-deterministic interleaving via threads on a unique processor. We expect that such verification may require the development of new algorithms and specifications, adapted for the proof of a concurrent version, in particular for the execution on multi-core processors.

Future work will also include an extension of the verification to complex mixed software and hardware designs in order to avoid that a hardware failure alters the expected behavior of a verified hypervisor.

Whatever particular verification technique is used, formal verification of a microkernel or a hypervisor represents a great effort and remains valid only for a particular version being verified. Therefore, any evolution of the software requires new verification. To allow industrial usage of formally verified system software in a real-life environment, the verification of a new version should require only a limited effort, without carrying out a new specification and proof of the whole system. Another important future work direction is developing formal verification methodologies for modular proof such that any evolution has a clearly defined, limited impact on the verification.

V. Conclusion

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 component 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 and Scala compilers, the JVM implementation. The effort for verify the whole stack is not at all within the reach of a single team.

[74] estimated the effort for developping a standard library for a mathematical proof checker to 140 man-years. More work is needed to be able to give an estimate of the needed effort for verifying a cloud environment, but we think that it would be of the same order.

Ongoing and future technical work includes both ends of the stack: verification of MapReduce applications (by constructive methods), verification of Hama applications (by a method that extends [75]) and verification of parts of the Anaxagoros hypervisor.

References

[1] X. Leroy, "Verified squared: does critical software deserve verified tools?" in *Symposium on Principles of Programming Languages (POPL)*, T. Ball and M. Sagiv, Eds. ACM, 2011, pp. 1–2.

[2] J.-R. Abrial, *The B-Book*. Cambridge University Press, 1996.

[3] J.-C. Filliâtre, "Verifying two lines of C with Why3: an exercise in program verification," in *Verified Software: Theories, Tools and Experiments (VSTTE)*, ser. LNCS, vol. 7152. Springer, January 2012, pp. 83–97.

[4] The Coq Development Team, "The Coq Proof Assistant," http://coq.inria.fr.

[5] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*. Springer, 2004.

[6] X. Leroy, "Formal verification of a realistic compiler," *CACM*, vol. 52, no. 7, pp. 107–115, 2009.

[7] P. Letouzey, "Coq Extraction, an Overview," in *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, ser. LNCS 5028, A. Beckmann, C. Dimitracopoulos, and B. Löwe, Eds. Springer, 2008.

[8] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml System release 3.12," http://caml.inria.fr, 2010.

[9] E. Chailloux, P. Manoury, and B. Pagano, *Développement d'applications avec Objective Caml*. O'Reilly France, 2000, freely available in english at http://caml.inria.fr/oreilly-book.

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. USENIX Association, 2004, pp. 137–150.

[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *International conference on Management of Data*, ser. SIGMOD'10. ACM, 2010, pp. 135–146.

[12] T. White, *Hadoop – The Definitive Guide*, 2nd ed. O'Reilly, 2010.

[13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*, Java SE 7 ed. Oracle, 2011.

[14] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 2nd ed. Artima, 2010.

[15] Oracle, "Oracle JRockit Virtual Edition," http://docs.oracle.com/cd/E23009_01/index.htm, 2012.

[16] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989, available at http://homepages.inf.ed.ac.uk/mic/Pubs.

[17] S. Pelagatti, *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[18] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software, Practrice & Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[19] L. G. Valiant, "A bridging model for parallel computation," *Comm. of the ACM*, vol. 33, no. 8, p. 103, 1990.

[20] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maen, "HAMA: An Efficient Matrix Computation with the Mapreduce Framework," in *2nd International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2010, pp. 721 –726.

[21] R. H. Bisseling, *Parallel Scientific Computation*. Oxford University Press, 2004.

[22] R. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM*, vol. 24, pp. 44–67, 1977.

[23] M. S. Feather, "A survey and classification of some program transformation approaches and techniques," in *The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation*. North-Holland Publishing Co., 1987, pp. 165–195.

[24] R. Bird, "An introduction to the theory of lists," in *Logic of Programming and Calculi of Discrete Design*, M. Broy, Ed. Springer-Verlag, 1987, pp. 3–42.

[25] J. Jeuring, "Theories for algorithm calculation," Ph.D. dissertation, Rijksuniversiteit Utrecht, The Netherlands, 1993.

[26] R. Bird and O. de Moor, *Algebra of Programming*. Prentice Hall, 1996.

[27] M. Cole, "Parallel Programming with List Homomorphisms," *Parallel Processing Letters*, vol. 5, no. 2, pp. 191–203, 1995.

[28] Z. Hu, M. Takeichi, and W.-N. Chin, "Parallelization in calculational forms," in *POPL'98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1998, pp. 316–328.

[29] A. Morihata, K. Matsuzaki, Z. H. Hu, and M. Takeichi, "The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer," in *Symposium on Principles of Programming Languages (POPL)*, Z. Shao and B. C. Pierce, Eds. ACM Press, 2009, pp. 177–185.

[30] R. S. Bird, "Lectures on constructive functional programming," in *Constructive Methods in Computing Science*, ser. NATO ASI, M. Broy, Ed., no. 55. Marktoberdorf, BRD: Springer, 1989.

[31] ——, "Functional Pearl: A program to solve Sudoku," *JFP*, vol. 16, no. 6, pp. 671–679, 2006.

[32] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2007.

[33] B. O'Sullivan, D. Stewart, and J. Goerzen, *Real World Haskell*. O'Reilly, 2008.

[34] 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.

[35] N. Javed and F. Loulergue, "Parallel Programming and Performance Predictability with Orléans Skeleton Library," in *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2011, pp. 257–263.

[36] J. Tesson, "Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels," Ph.D. dissertation, LIFO, University of Orléans, November 2011. [Online]. Available: http://hal.archives-ouvertes.fr/tel-00660554/en/

[37] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson, "Systematic Development of Correct Bulk Synchronous Parallel Programs," in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2010, pp. 334–340.

[38] F. Loulergue, F. Gava, and D. Billiet, "Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction," in *International Conference on Computational Science (ICCS)*, ser. LNCS 3515, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Springer, 2005, pp. 1046–1054.

[39] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski, "Bulk Synchronous Parallel ML with Exceptions," *Future Generation Computer Systems*, vol. 26, pp. 486–490, 2010.

[40] R. Lämmel, "Google's MapReduce programming model – Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, 2008.

[41] Y. Liu, Z. Hu, and K. Matsuzaki, "Towards Systematic Parallel Programming over MapReduce," in *Euro-Par 2011 Parallel Processing*, ser. LNCS, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin / Heidelberg, 2011, vol. 6853, pp. 39–50.

[42] K. Emoto, S. Fischer, and Z. Hu, "Generate, Test, and Aggregate – A Calculation-based Framework for Systematic Parallel Programming with MapReduce," in *ESOP*, ser. LNCS, vol. 7211. Springer, 2012, pp. 254–273.

[43] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS 2283. Springer, 2002.

[44] Y. Imai and J. Fan, "Coq2Scala," http://proofcafe.org/wiki/en/Coq2Scala, February 2012.

[45] C. A. R. Hoare, "An axiomatic basis for computer programmation," *Communication of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[46] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," in *Formal Methods for Components and Objects (FMCO)*, 2005, pp. 364–387.

[47] J.-C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification," in *19th International Conference on Computer Aided Verification*, ser. LNCS, W. Damm and H. Hermanns, Eds. Springer, 2007.

[48] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer, "CC(X): Semantic Combination of Congruence Closure with Solvable Theories," *Electronic Notes in Theoretical Computer Science*, vol. 198, no. 2, pp. 51–69, 2008.

[49] P. Herms, C. Marché, and B. Monate, "A certified multi-prover verification condition generator," in *Verified Software: Theories, Tools, Experiments, (VSTTE)*, ser. LNCS, R. Joshi, P. Müller, and A. Podelski, Eds. Springer, 2012.

[50] K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya, "Using Coq in specification and program extraction of Hadoop MapReduce applications," in *Proceedings of the 9th international conference on Software engineering and formal methods*, ser. SEFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 350–365.

[51] F. Gava, "Formal Proofs of Functional BSP Programs," *Parallel Processing Letters*, vol. 13, no. 3, pp. 365–376, 2003.

[52] J. Tesson and F. Loulergue, "A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation," in *International Conference on Computational Science (ICCS)*, ser. Procedia Computer Science. Elsevier, 2011, pp. 36–45.

[53] G. Stolpmann, "The Plasma Project," http://plasma.camlcity.org, November 2011.

[54] C. Haack, M. Huisman, and C. Hurlin, "Permission-Based Separation Logic for Multithreaded Java Programs," *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, vol. 15, pp. 13–23, 2011.

[55] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[56] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *Journal of Automated Reasoning*, vol. 41, no. 1, pp. 1–31, 2008.

[57] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[58] G. Klein and T. Nipkow, "A machine-checked model for a Java-like language, virtual machine, and compiler," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 4, pp. 619–695, 2006.

[59] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*, Java SE 7 ed. Oracle, 2011.

[60] P. Altherr, "A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[61] N. Torp-Smith, L. Birkedal, and J. C. Reynolds, "Local reasoning about a copying garbage collector," *TOPLAS*, vol. 30, no. 4, 2008.

[62] M. O. Myreen, "Verified just-in-time compiler on x86," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL'10. New York, NY, USA: ACM, 2010, pp. 107–118.

[63] M. Lemerre, V. David, and G. Vidal-Naquet, "A communication mechanism for resource isolation," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, ser. IIES'09. New York, NY, USA: ACM, 2009, pp. 1–6.

[64] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques, "Method and Tools for Mixed-Criticality Real-Time Applications within PharOS," in *Proceedings of AMICS 2011: 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*, 2011.

[65] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole, "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of SOSP '95*, 1995, pp. 251–266.

[66] J. H. Saltzer, "Protection and the control of information sharing in multics," *Commun. ACM*, vol. 17, pp. 388–402, July 1974.

[67] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*. ACM, 2009, pp. 207–220.

[68] The Common Criteria Recognition Arrangement, http://www.commoncriteriaportal.org.

[69] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban, "Pervasive Verification of an OS Microkernel," in *Verified Software: Theories, Tools, Experiments*, ser. LNCS, G. Leavens, P. O'Hearn, and S. Rajamani, Eds. Springer Berlin / Heidelberg, 2010, vol. 6217, pp. 71–85.

[70] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *Verified software: theories, tools, experiments (VSTTE)*. Springer, 2010, pp. 40–54.

[71] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, , and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, vol. 5674, 2009, pp. 23–42.

[72] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language*, Feb. 2011, http://frama-c.cea.fr/acsl.html.

[73] L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski, *Frama-C User Manual*, Oct. 2011, http://frama-c.com.

[74] F. Wiedijk, "Estimating the Cost of a Standard Library for a Mathematical Proof Checker," http://www.cs.ru.nl/~freek/notes/mathstdlib2.pdf.

[75] J. Fortin and F. Gava, "BSP-WHY: an intermediate language for deductive verification of BSP programs," in *4th workshop on High-Level Parallel Programming and applications (HLPP)*. ACM, 2010, pp. 35–44.