An Algorithmic Approach to Compositional Verification of Sequential Programs with Procedures: An Overview

Dilian Gurov[1], Marieke Huisman[2], and Christoph Sprenger[3]

[1] Royal Institute of Technology, Stockholm, Sweden
[2] INRIA Sophia Antipolis, France
[3] ETH Zurich, Switzerland

Abstract This extended abstract gives an overview of an algorithmic technique for compositional verification of sequential programs with procedures. The technique addresses control flow safety properties expressed in a suitable temporal logic, and is based on a maximal model construction for algorithmically deciding the correctness of property decomposition. Here, we summarize the approach and discuss its main ingredients and contributions. We also discuss related and future work, and conclude with references to the relevant publications, where the technical details are presented.

1 Motivation

Over the last years, computer systems have become increasingly dynamic: they are composed of various communicating components that can join the system or be put together dynamically. Typical examples are mobile smart devices (mobile phones, smart cards, television set top boxes, PDAs *etc.*) and dynamically reconfiguring distributed systems. When allowing the dynamic addition of new components, one wishes to ensure that this will not have any negative impact on the global behaviour of the system. In particular when the system contains privacy–sensitive information, as is for example the case for smart cards containing health care information or electronic purses, strong security guarantees are required. With the acceptance of evaluation schemes such as Common Criteria[1], industry has come to realise that the way to achieve such high guarantees is to adopt the use of formal methods in industrial practice.

The techniques that we have developed are applicable in any context concerned with inter-procedural control-flow properties of components communicating via procedure calls. Interesting properties of such components include for example type safety, memory consumption, and illicit data or control flow. We concentrate on the last category of properties. More precisely, we study sequential (*i.e.,* single-threaded) programs and propose a specification and verification

[1] See `http://www.commoncriteria.org`.

method for safety properties of inter-procedural control flow, *i.e.,* properties describing safe sequences of procedure invocations. Typical examples of control-flow safety properties are: "m_1 never calls m_2", "m_1 is never called when m_2 is called", "m_1 is only called after m_2 is called", and "m_1 is only called from within m_2" (see [14] for a formalisation).

So far, most research on formal verification in this area has focused on the correctness or security of a single program component (*e.g.,* [17,8,3]). However, in the context of mobile code we also need techniques to support verification of systems for which it is not known in advance what its components will be. In such situations one needs *compositional verification techniques*, that is techniques where one states minimal requirements for the components that can become available later, and then verifies (at loading time) that the components actually respect these requirements. Only then, existing components can safely communicate with new components, without corrupting the correctness or security of the whole system. In particular, such techniques can support the secure post-issuance loading of new applications onto smart devices. Such a compositional verification technique should not only be sound, but also complete, in the sense that false negatives are avoided, *i.e.,* components that are actually secure should not be rejected[2].

2 Approach

Our verification method is *compositional*: it allows us to verify global guarantees of a system even if the implementations of some components are not yet available at verification time. This is achieved by abstracting the missing components by logical assumptions. These assumptions can be verified later, when the implementations become available. Such a verification approach is embodied by the following proof principle:

$$\frac{\models A : \phi \qquad X : \phi \models X \otimes B : \psi}{\models A \otimes B : \psi}$$

where A and B are components, and X is a component variable. This principle reduces the problem of showing that the composition of components A and B satisfies ψ, where the implementation of A is not yet known, to three tasks:

1. decompose the global property ψ by finding a suitable local property ϕ of component A,
2. prove *correctness* of the decomposition, *i.e.*, verify that for *any* component X satisfying ϕ, X composed with B satisfies ψ (second premise), and

[2] Completeness is also crucial to avoid typical social engineering attacks, where the device user gets so frustrated with the system repeatedly rejecting new components, that he/she will simply accept all, without actually inspecting whether they passed verification or not.

3. when the implementation of A becomes available, verify that it satisfies the local property ϕ (first premise).

Notice that this rule can be applied repeatedly, to replace several components by assumptions.

The compositionality of the method supports different scenarios for secure configuration of components on a device (or platform), where the tasks above can potentially be delegated to different authorities. In one such scenario, the device issuer (or platform provider) specifies both the global guarantee (*e.g.,* a security policy) and the local assumptions, and verifies – using the techniques described in this paper – that the decomposition is correct, meaning that the local specification is sufficient to establish the global specification. Each time a new component is to be added (*i.e.*, loaded on the device), an algorithm provided by the device issuer checks whether the component implementation satisfies the required specification. An alternative scenario is that the device issuer only provides the global guarantee (and local assumptions for its own components), and leaves it to the component provider to come up with an appropriate local specification for each component to be added. As in the previous scenario, an algorithm provided by the device issuer checks the component against the local specification upon loading, but now also the property decomposition needs to be verified at loading time, potentially on–device.

Task (1) above is a manual one and requires insight into the system, while the other two can be automated. We show how Task (2) and Task (3) can be algorithmically reduced to problems for which standard algorithmic techniques exist [22].

The approach that we take to handle Task (2) is inspired by the pioneering work on automatic modular verification by Grumberg and Long [19]. To check whether $X : \phi \models X \otimes B : \psi$ holds we replace X by a *maximal model* $\theta(\phi)$ and then verify $\models \theta(\phi) \otimes B : \psi$ algorithmically. The maximal model $\theta(\phi)$ represents all models satisfying ϕ in the sense that it *simulates* exactly those models and thus satisfies precisely the properties enjoyed by all these models. For this technique to be sound and applicable it is required that maximal models exist for the chosen logic and simulation relation, $\otimes$ preserves simulation, and logical properties are preserved by simulation. In earlier work [7], we explored deductive verification of correctness of decompositions based on a proof system. The logic considered there was more expressive, but the interactive nature of the approach required considerable time and expertise from the user, rendering the approach less preferable in many situations as compared to algorithmic solutions like the one presented here.

We are interested in *safety properties* of both the *structure* and the *behaviour* of programs. Since the same behaviour can be brought about by different structures, a behavioural property language allows properties to be expressed in a more abstract fashion. However, as a rule, behavioural properties require computationally more expensive verification techniques. Still, they can often be (equivalently) reformulated on the structural level, with the advantage of allowing more efficient verification. To support both kinds of properties, we distinguish between

a structural and a behavioural level of programs. Both structure and behaviour are cast via the abstract notion of *model* (or labelled Kripke structure). Then, structural properties are interpreted over the (finite-state) control-flow graphs themselves, while behavioural properties are interpreted over the (infinite-state) behaviours induced by the structures. The logic we employ to express such properties is a modal logic with box modalities and simultaneous greatest fixed points (written in equational form), which is expressively equivalent to the fragment of the modal mu-calculus with box modalities and greatest fixed points only [25]. The fragment is known to be adequate for expressing safety properties (*cf.* [11]). Because of the close relationship between logical satisfaction and *simulation* between models, and the compositional properties of simulation, this logic, which for convenience we term *simulation logic*, is particularly suitable for compositional verification via maximal models. We instantiate simulation logic and simulation at both the structural and the behavioural levels.

To handle Task (3) for programs with private procedures, we define the notion of *interface behaviour*, which gives an abstract view of the program behaviour by focusing on the public procedures. In addition, we define an *inlining transformation* that recursively inlines all calls to private procedures. This transformation over-approximates the interface behaviour, thus reducing the task to showing that the inlined program respects property ϕ. For the latter, we apply standard algorithmic verification techniques.

3 Contributions

The main contribution of our work is a sound and complete compositional verification principle for sequential programs with procedures, for properties expressed in simulation logic, and its adaptation to programs with private procedures. In more detail, the contributions are as follows.

Program Model. Most of the existing work on compositional model checking focuses on the verification of parallel compositions of finite-state processes. We extend compositional model checking to an important class of infinite-state programs, namely sequential programs with procedures. In our papers, we refer to programs as *applets* and to procedures as *methods*, but we would like to stress that our technique is applicable to many different kinds of programs with procedures. We represent applets as collections of method control-flow graphs equipped with *interfaces* of provided and required methods. Applet *composition* forms the disjoint union of the respective collections of method graphs and allows the composed applets to communicate via method invocation. Applets correspond to a subclass of pushdown processes, with potentially infinite-state behaviour (*cf.* [13]).

Maximal Model Construction. We establish a *logical characterisation* of the standard notion of simulation between models and, vice versa, a behavioural characterisation of logical satisfaction in terms of *maximal models*. In particular, we

have developed a novel maximal model construction, consisting of a step-wise transformation of the formula into a semantically equivalent normal form, which is isomorphic to a maximal model for the formula. To the best of our knowledge, this is the first maximal model construction for (a variant of) the modal μ-calculus, which includes the full expressive power of simultaneous greatest fixed points.

Maximal Applet Construction. When tailoring the maximal model technique to applets, we require that the maximal model for a given property is itself an applet. This is necessary for *completeness* of the technique. Since the verification of $\models \theta(\phi) \otimes B : \psi$ is decidable in our setup, completeness guarantees that if the verification of the correctness of decomposition fails, there is indeed an *applet* F among the set of models such that F satisfies ϕ but $F \otimes B$ does not satisfy ψ. Completeness is thus essential in that it eliminates the possibility of false negatives. Therefore, in case $\models \theta(\phi) \otimes B : \psi$ fails, we know that we have to strengthen ϕ and iterate the process.
To adapt the maximal model technique to structural properties, we first give a logical characterisation of interfaces by defining, for a given interface I a structural formula ϕ_I which is satisfied exactly by those models representing applet structures with this interface, and then define the *maximal applet* for a given interface I and structural property ϕ by $\theta_I(\phi) = \theta(\phi_I \wedge \phi)$. Since $\theta(\phi_I \wedge \phi)$ satisfies both ϕ and ϕ_I, this guarantees that the resulting maximal model is indeed an applet structure with interface I satisfying the structural formula ϕ.
However, for behavioural properties there is in general no unique maximal applet: different applets, incomparable by simulation, might exist that satisfy the same property. It is ongoing work to investigate under what conditions and how this collection of maximal applets can be characterised exactly. Preliminary results in this direction are available from [21].

Compositional Verification. Our characterisation results, together with results linking the structural and behavioural levels, give rise to a *compositional verification principle* of the shape suggested above, where the global guarantee can be either structural or behavioural, but the local assumptions are always structural. We establish the *soundness* and *completeness* of the principle, and adapt existing algorithmic techniques for dealing with the resulting verification sub-tasks.

Interface Abstraction. To address programs with private methods, we extend our compositional verification method to *interface properties* of applets. This extension is necessary to keep the verification time for real-life applications acceptable (*i.e.,* within a few seconds). To abstract from internal, private behaviour we define an *abstraction* which reduces the set of methods of a given applet to the set of its public methods, while over-approximating the interface behaviour of the applet. This abstraction is based on *inlining* of private methods. We define the notion of *interface behaviour*, and show the abstraction to be *sound* with respect to public interface properties: every property that holds for the interface

behaviour of the inlined applet (which coincides with its behaviour since it has no private behaviour) also holds for the interface behaviour of the original applet. Since the abstraction transformation can introduce new interface behaviours, completeness, on the other hand, does not hold in general. However, for the case when the concrete implementation is *last-call recursive* (that is, recursive calls are not followed in the control-flow graph by any other method calls[3]), the abstraction technique is *complete* with respect to observable interface properties: if such a property does not hold of the inlined applet it does not hold of the original applet either.

Tool support and real-life case study. To support our compositional verification technique, we have developed a tool set. This tool set integrates our own implementations in Ocaml of the maximal applet construction and the inlining algorithm with an implementation of a model extractor, build on top of the SOOT framework [34], and a number of external model checking tools. We have validated this tool set on an industrial case study, namely an electronic purse smart card applet for which we have verified the absence of certain illicit control flows between *Purse* and *Loyalty* applets. In particular, we ensured that different *Loyalty* applets on the card cannot communicate information about the transaction log table – that is needed to correctly compute the points in the loyalty program – among themselves, instead they all need to register (and pay) to get this information directly from the *Purse*. In this case study, the inlining technique proved to be an essential ingredient that enabled the compositional verification of the otherwise too large model.

Note that our contributions span the complete spectrum from the theoretical underpinnings of the compositional applet verification technique (our principal contribution) to its support by a tool set and its application to an industrial case study.

4 Related Work

The techniques and tools that we have developed to support compositional verification of sequential procedures are related to several different research areas.

Program Model. The program model used in the present paper has been inspired by the work of Besson *et al.* [8], who verify stack properties for Java programs. Typically, the behaviour of programs with recursion is modelled as Pushdown Automata, as *e.g.,* in [17].

Recursive state machines were introduced by Alur *et al.* [3] as a formalism capable of modelling the control flow of sequential imperative programs containing recursive procedure calls. This program model is closely related to our own,

[3] This notion is a generalisation of the notion of *tail recursiveness*, where recursive calls are the last statements of their methods. In practice, for industrial code it is very common to be last-call recursive.

but is finer in that calls and returns relate individual entry and return nodes, thus allowing the effect of data to be modelled. The authors develop efficient algorithms for (global) model checking of recursive state machines against LTL and CTL* properties, and investigate their complexity.

Temporal Logic. Related to the above program models is the temporal logic of calls and returns CaRet proposed by Alur *et al.* [4]. This logic allows to specify properties in terms of method calls and returns, thus increasing the expressiveness of temporal logic while retaining decidability of model checking. A special verification strategy is defined, that is able to "jump over" internal computations. An extension of this logic was recently presented by Alur *et al.* [2]. Among other modalities, it introduces the useful "within" modality, which is not expressible in simulation logic. While these logics may be more adequate than simulation logic for specifying behavioural properties of programs with procedures, they would (arguably) require more involved techniques for compositional verification.

Compositional Verification. There is a wealth of methods for compositional verification of concurrent programs, most notably assumption/commitment based reasoning about processes with synchronous message passing, and the rely/guarantee method for shared-variable concurrency. A systematic overview of these and related proof methods, some of which have been adapted to support algorithmic verification is given by De Roever *et al.* [32]. However, these techniques do not address programs with recursive procedures.

Laster and Grumberg [28] present a compositional method for sequential programs written in a high-level While language (without procedures). Their technique partitions the program text into a sequence of sequentially composed subprograms, which can be model checked individually using assumptions on the properties holding at the cut points.

Alur and Grosu [5] present an assume-guarantee style compositional verification principle for a hierarchic extension of reactive state machines. However, their approach does not address programs with recursion.

Ly [29] also proposes a compositional method for deciding control-flow properties of procedural programs based on local structural assumptions and global behavioural guarantees. He generalises our decidability results to monadic second-order logic for programs whose control-flow graphs have a bounded tree-width. To the best of our knowledge, so far this approach has not been implemented in a tool.

The method of partial model checking introduced by Andersen [6] is based on a reduction procedure that removes the top-level operator from a process algebra term and computes a new property for the reduced term. To verify that the product $P \times Q$ of two processes has some property ϕ, the reduction "divides" the property ϕ by Q to yield ϕ/Q, which can be effectively computed only if Q is finite.

Maximal Models for Compositional Verification. The original maximal model technique by Grumberg and Long [19] was designed for ACTL, the universal frag-

ment of CTL, and later extended to ACTL*, the universal fragment of CTL*, by Kupferman and Vardi [26]. These works study synchronous parallel compositions of sequential processes under fairness assumptions. Since we are interested in safety properties of sequential programs, we do not need to add fairness to our models. Simulation logic and ACTL* are expressively incomparable: liveness properties such as GFp ("infinitely often p") are expressible in ACTL*, but not in simulation logic, while the μ-calculus formula $\nu X.\, p \wedge [-][-]X$ ("p holds on every other level of the computation tree") is easily translated to simulation logic (which is in equational form), but not expressible in ACTL*. Our transformational approach to the maximal model construction is closer to an implementation than the automata-theoretic constructions in the cited papers, since it already includes certain optimisations, *e.g.,* removal of duplicate and unreachable equations.

Characterisation results connecting logics and behavioural preorders similar to ours are described by Larsen and Boudol [12] (see also [27]), who construct maximal models in the form of modal transition systems *w.r.t.* the refinement preorder for Hennessy-Milner logic (HML) [23]. Simulation logic and HML are expressively incomparable: existential properties are not expressible in simulation logic, while co-recursive properties (such as invariants) are not expressible in HML. Since HML does not include fixed points, the constructed maximal models are essentially finite forests. Apart from the absence of diamond modalities in simulation logic, our construction can be seen as an extension of Larsen and Boudol's with greatest fixed points. The extension of HML with greatest fixed points (or, equivalently, simulation logic with diamond modalities) requires more general models than modal transition systems: a finite maximal modal transition system does not exist for all formulae of this logic ([15], see also [16]). Bouajjani *et al.* [11] define maximal models for a co-recursive modal logic expressing safety properties. Their logic has an expressive power similar to ours, but is somewhat less standard as it includes a connective corresponding to non-deterministic choice.

A more recent application of the maximal model technique is presented by Goldman and Katz [18] in the context of modular verification of *aspects*. While close in spirit to our verification principle, the principle presented by the authors is for a more complicated composition operator. The principle is based on the maximal model of the aspect property (which is not necessary a legal aspect behaviour) and is therefore sound, but not complete.

5 Future work

The program model which forms the basis for our analyses is rather abstract. We are currently investigating how to extend our techniques to finer program models. In particular, we are considering program models capturing multi–threading and exceptions. Our compositional verification principle remains valid, as long as the notions of structure and behaviour (and the corresponding notions of simulation

and logic) can be extended so that the necessary technical conditions still apply. However, the verification problem for the global behavioural property becomes undecidable in the presence of multi–threading [31] (when considering the same primitives as in *e.g.,* Java), thus appropriate abstraction techniques have to be employed for this task (as proposed in *e.g.,* [10,9,30]). A further extension of significant interest is adding data to the program model, so that a more precise control flow can be modelled, and properties over data can be specified. This requires again the use of appropriate abstractions in order to retain decidability of the verification problems.

In principle, our verification technique can be extended to more powerful logics, for example to the full modal μ-calculus. However, adding diamond modalities and least fixed-point recursion to the logic requires a more general notion of model (and hence applet structures and behaviours) in the framework; for example, see [16,1] for such models and corresponding maximal model constructions.

Further we are investigating under what restrictions one can construct maximal applets for behavioural properties, thus extending the method to deal with local behavioural properties. The approach we take is to define a translation from behavioural properties into collections of structural properties, such that any applet that is simulated by a maximal applet for one of the structural properties satisfies the original behavioural one. Preliminary results in this direction are available from [21].

6 Further Reading

This paper contains (slightly modified) fragments from a long overview paper, describing our work [22]. The overview paper has the following contents. First, it presents the theoretical foundation for our work: it defines the models and logic that we consider, together with appropriate notions of simulation and satisfaction. Next, it presents our novel maximal model construction, and shows how logical satisfaction of a formula is equivalent to simulation by the corresponding maximal model. Then it discusses how our results instantiate to applets, at structural and at behavioural level, and presents the compositional verification principle. Next, the inlining abstraction that we use to be able to verify interface properties over applets with private methods, is presented. Finally, it illustrates how our approach is implemented as a tool set and is applied to an industrial case study.

Several results have been presented earlier. The maximal model construction and compositional verification principle are presented in [33]. The abstraction technique for applets with private methods is presented in [20]. The case study was presented in [24], but without taking the difference between public and private methods into account.

References

1. I. Aktug and D. Gurov. State space representation for verification of open systems. In *Algebraic Methodology And Software Technology (AMAST'06)*, volume 4019 of *LNCS*, pages 5–20. Springer Verlag, 2006.
2. R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Proc. LICS 2007*, 2007.
3. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM TOPLAS*, 27:786–818, 2005.
4. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Analysis and Construction of Software, TACAS'04*, number 2998 in LNCS, pages 467–481. Springer Verlag, 2004.
5. R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. *ACM TOPLAS*, 26:339–360, 2004.
6. H.R. Andersen. Partial model checking (extended abstract). In *Logic in Computer Science (LICS 95)*, pages 398–407. IEEE Computer Society Press, 1995.
7. G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering, FASE'02*, number 2306 in LNCS, pages 15–32. Springer Verlag, 2002.
8. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. of Computer Security*, 9(3):217–250, 2001.
9. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejček. Reachability analysis of multithreaded software with asynchronous communication. In R. Ramanujam and S. Sen, editors, *Proceedings of FSTTCS 2005*, number 3821 in LNCS. Springer Verlag, 2005.
10. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Principles of Programming Languages, (POPL '03)*, 2003.
11. A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Automata, Languages and Programming (ICALP 91)*, volume 501 of *lncs*, pages 76–92. springer, July 1991.
12. G. Boudol and K. Larsen. Graphical versus logical specifications. *Theoretical Computer Science*, 106:3–20, 1992.
13. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.
14. G. Chugunov, L.-å. Fredlund, and D. Gurov. Model checking of multi-applet JavaCard applications. In *CARDIS'02*, pages 87–95. USENIX Publications, 2002.
15. D. Dams and K.S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 335–344, Los Alamitos, CA, 2004. IEEE Computer Society Press.
16. D. Dams and K.S. Namjoshi. Automata as abstractions. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385, pages 216–232. springer, 2005.
17. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification (CAV '00)*, number 1855 in LNCS, pages 232–247. Springer Verlag, 2000.

18. M. Goldman and S. Katz. MAVEN: Modular aspect verification. In O. Grumberg and M. Huth, editors, *TACAS 2007*, number 4424 in LNCS, pages 308–322. Springer Verlag, 2007.
19. O. Grumberg and D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.
20. D. Gurov and M. Huisman. Interface abstraction for compositional verification. In B. Aichernig and B. Beckert, editors, *Proc. SEFM'05*, pages 414–423, Koblenz, Germany, September 2005. IEEE Computer Society.
21. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. Technical Report TRITA-CSC-TCS 2007:3, KTH Royal Institute of Technology, Stockholm, 2007. Available at `http://www.csc.kth.se/~dilian/Papers/techrep-07-3.pdf`.
22. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*. Conditionally accepted, preliminary version available at `http://www.csc.kth.se/~dilian/Papers/cvspp06.pdf`.
23. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
24. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, FASE'04*, number 2984 in LNCS, pages 84–98. Springer Verlag, 2004.
25. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
26. O. Kupferman and M. Vardi. An automata-theoretic approach to modular model checking. *ACM TOPLAS*, 22(1):87–128, 2000.
27. K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, number 407 in LNCS. Springer Verlag, 1989.
28. K. Laster and O. Grumberg. Modular model checking of software. In *Tools and Algorithms for the Analysis and Construction of Software, TACAS'98*, LNCS. Springer Verlag, 1998.
29. O. Ly. Compositional verification: Decidability issues using graph substitutions. In *Proceedings of the 29th Mathematical Foundations of Computer Science, MFCS 2004*, volume 3153 of *LNCS*, pages 537–549. Springer Verlag, 2004.
30. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, LNCS. Springer Verlag, 2005.
31. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22(2):416–430, 2000.
32. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.
33. C. Sprenger, D. Gurov, and M. Huisman. Compositional verification for secure loading of smart card applets. In *Formal Methods and Models for Co-Design (Memocode 2004)*, pages 211–222. IEEE, 2004.
34. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.