English Introduction

Nowadays, one of the most complex parts of critical systems is the system software. It is found in applications ranging from airplane navigation systems [20] to train control systems [6], as well as nuclear power plants or space shuttles, among many others. Due to the nature of this kind of system, where failures are not only measured in financial terms, but also in potential loss or damage to human life, the software used must be totally reliable. Moreover, these kind of systems are concurrent systems where the application of traditional debugging and testing techniques is very complex. In *software* engineering, tools for test case generation (testing) can only cover a small part of the space of states needed to be explored, and the correctness and reliability of the whole system cannot be ensured. On the other hand, formal verification tools can provide methods that perform a more exhaustive analysis. Among these techniques, we can highlight theorems prover or the verification of models also known as MODEL CHECKING [17] .

Today, MODEL CHECKING is one of the most commonly used formal techniques used to establish that a system is adapted to particular requisites, normally specified in some logic like LTL [46] , CTL [7, 15, 23] or μ-calculus [36]. Research in this area has produced a large number of tools, in the academic environment (SPIN [32, 33] , CADP [27, 28] ,NUSMV [14]), as well as in the business one, oriented to specification languages, which are known as *formal description techniques* (FDT), such as PROMELA, SDL or LOTOS. The adaptation of MODEL CHECKING techniques to programming languages has become a good way to improve the quality of both concurrent and critical systems. In the last decade, this method has been adapted to real programming languages, such as C, C++ or JAVA. These *software* MODEL CHECKING tools are based on the same state space exploration algorithms designed for the formal description techniques, and some of them incorporate additional features that are not included in the specificaticion languages, such as pointers or dynamic memory management.

In this thesis, we provide a methodology to obtain models from programs that make use of functions which are external to the language (e.g. well defined APIs). Moreover, we present two mechanisms that make possible the verification of programs that use dynamic memory and allow the analysis of properties over dynamic structures.

Motivation

Extending explicit MODEL CHECKING to deal with programming languages involves certain problems. On the one hand, we have to choose a formalism to verify the code and to manage the external functions present in the program; on the other hand, if the program uses dynamic memory, we have to provide an internal representation of dynamic structures (the *heap*), and the references to it, and we must also consider how to specify and verify the properties of these dynamic structures. In the following, we describe some related work in these areas.

Software verification and external functions

In the context of the verification of concurrents programs using MODEL CHECKING there are two different approaches. The first is the model extraction, which consists of translating the program that is going to be analyzed into a formal description technique valid for some existing MODEL CHECKER (see FEAVER [42] for C, and JPF1 [29] and BANDERA [18] for JAVA) . The translation usually involves reducing the program detail (abstraction) in such a way that the final model only contains relevant information on the properties that will be analyzed. That the abstraction is correct with regard to these properties is a critical aspect that must be considered. One important point of this approach is that less effort is required to get the tools. It is also important that future optimizations in the MODEL CHECKERS can be used without being implemented.

The second approach consists of implementing “specific tools” that are designed to analyze a particular programming language (see JPF [37, 50] for JAVA, and SLAM [4, 5] , CMC [44] or BLAST [9, 31] for the C language). These tools do not need the translation step and this makes it easier to come back to the code from a particular error. However, they require more time to be spent on development and resources than the tools that use an external tool to do the program analysis.

Regarding the use of external functions to the language, several different approaches are possible. So, the tool CMC requires that the user provides an enviroment including a minimal functionallity of the external functions present in the code. In JPF, the system verifies functions that do not cause side effects, or those that involve undefined structures such as the communication buffers belonging to the operating system. By default, the tool includes a black list of packages that will not be verified. This list is composed by: $java.*$, $javax.*$ y $sun.*$ containing some packages like, for example, SOCKETS. Otherwise, SLAM, focusses in *driver* verification using the *windows driver model* (WDM) interface.

Heap representation

From the implementation point of view, the problem is how to deal with the internal representation of the states during exploration of potential program behavior. Model checking algorithms are optimized to consider global states with a fixed structure and length, and should be modified in order to deal with states having different configurations that depend on operations to allocate and free memory. There are some proposals describing representations of the state for C [21, 44] and JAVA [37] .

The most natural approach for dealing with dynamic structures is to allocate a heap for every process in the state vector. In this way, the state vector contains all the static and dynamic variables for all the processes, although a high price has to be paid in terms of memory use. This is the approach initially followed by the CMC tool , which is able to do model checking of C and C++ programs. In order to avoid using so much memory, CMC uses a hash table where only a signature of the state is stored. This kind of compression of the state vector produces partial verification. It also uses a local heap for each process and it implements a mechanism to avoid checking unused parts of the allocated memory if they are not referenced from pointers.

Model checker BLAST focuses on verifying that programs contain no memory leaks. It works by transforming the source code to include assertions associated to the statements that manage dynamic memory. Errors are reported as counterexamples.

Tool dSPIN [21] extends SPIN with new PROMELA sentences and modifies the basic SPIN implementation. The language is extended with a notation to identify pointer variables, in such a way that the operations regarding pointers (assignment and comparison) are given different semantics which are context-dependent. Their behavior depends on the position (left or right) of the pointer variable in assignments and on the type of instruction on the other side. Internally, the tool uses an extensible vector state with a separate area for dynamic objects for

each state. This extensible state is linearized at every step of the model checking process in order to produce a representation compatible with the SPIN algorithms to perform matching, hashing and state compression. Apart from considering the heap in every state (at least partially), this linearization is a time consuming process; however, only the relevant information should be copied to the linear state using a canonical representation of the heap.

JPF, a Java oriented model checker, also considers the separation of static and dynamic parts of the state vector. Dynamic objects are stored as a global pool of values, and only the pool indexes are placed in the static part of the state vector, together with the static variables. This way of collapsing the state increases the time and memory needed to verify large examples due to backtracking (necessary to perform exhaustive exploration of the bytecode corresponding to the Java program). Therefore the authors also implement a reverse collapse method to manage the states.

Finally, BOGOR [48] is a framework to construct software model checkers. It is based on an internal language (BIR) that supports both dynamic creation of objects and garbage collection. The mechanisms to perform verification of dynamic memory are based on the dSPIN way of representing the heap.

Verifying properties over dynamic structures

The problem of analyzing properties of dynamic structures has been extensively studied in the literature, mainly from two complementary points of view. On the one hand, following the classic ideas of the static analysis theory, *shape analysis* automatically infers the shapes of structures allocated in the program heap [45]. Extracting this information is expensive and it usually involves analyzing data sharing or alias analysis [3], i.e., detecting when two different pointers refer to the same memory location. As usual, shape analysis may be used to optimize the code generated by the compiler prior to program execution. The most relevant reference in this area is the tool TVLA (*Three Valued Logic Analysis Engine*)) [10, 38, 49]. TVLA is an abstract interpreter that uses an extension of predicate abstraction (called canonical abstractions) to represent program states.

On the other hand, shape information may also be obtained from a specification given by the user. For instance, the notion of *Graph Types* introduced by [35] provides an abstract representation of the so-called *shape invariant* that is able to describe very complex dynamic structures. An algorithmic method may be used to check whether program data satisfy the shape invariant specified. In particular, the *Pointer Assertion Logic Engine* (PALE) [43] may express properties

described by graph types. Programs are annotated with these formulas and checked by the MONA tool, a theorem prover based on Hoare-triples. Similarly, shape types [24] and ADDS [30] (Abstract Description of Data Structures) define descriptions that may be considered as alternatives to Graph Types. The temporal evolution of the heap has also been described by the so-called *Evolution Temporal Logic* ETL [51]. The TVLA tool has been extended to support ETL specifications.

In recent years, and in the context of software model checking, new proposals have appeared. Bouajjani et al. [11] represent properties on pointers using finite-state automata. They employ regular model checking combined with abstraction to reason about the dynamic structures. In addition, they introduce the LBMP logic (logic of bad memory patterns) to describe undesirable behaviors [12]. Formulas are attached to the code in such a way that reachability analysis may be used to evaluate them. Furthermore, the GROOVE [34] tool focuses on making the dynamic structures available to check CTL formulas. Instead of a linear method, the authors use a graph representation, which is more suitable to implement efficient matching.

It is also worth noting that the introduction of the *separation logic* [47] has opened new lines of research in the subject. Separation logic is able to formally describe the shared information stored in the heap. Semantic rules for statements that manage pointers make it possible to prove properties of pointers using the traditional pre-post condition scheme. The combination of this logic with other proposals is a fruitful field of research [22].

Regarding the use of nesting or multiple dimensions in logic, several proposals have been made. The CaRet logic [2] extends LTL to reason about programs with nested functions, in such a way that the logic can match each call with the response returned. That work has been recently extended to reason on generic nested words [1]. LTL^{mem} [13] is also a multidimensional logic that uses LTL as the most external formalism. However, instead of using a model logic like CTL to reason about dynamic structures, it has an embedded separation logic. This logic has been mainly studied from the point of view of complexity for model checking, but no tool or experimental results are available as yet. A more general approach to defining spatio-temporal logic has also been proposed in [8]. However, this work does not consider dynamic structures or model checking.

It is necessary to define new property languages to express requirements and to reason about data structures created dynamically, such as linked lists. Model checkers usually employ variants of temporal logic to define properties about states and sequences of states. Atomic

propositions in these logics are related to the static variables in the program. When considering linked structures (with anonymous nodes), a new mechanism is needed to reason about them.

Contributions

In this thesis, we address two problems of verifying in a practical manner low-level code: the verification of programs that use of external functions to the language and the verification of dynamic data structures. In particular, for the first issue, we propose a methodology for the model extraction and its verification with regards to well defined APIs. For the second issue, we propose an efficient representation of the state vector during verification of C programs with dynamic structures and a new modal logic (MALTL) to reason about properties on dynamic data. Regarding the model extraction, the main contributions of this thesis are:

- A methodology for model extraction for applications that use well defined APIs. The propossed methodology is based on three main aspects: the analysis of the source code with respect to the control flow and program data, the formalization of the analyzed API, and the transformation of the code of the programing language to previously established formal description technique;
- The implementation of a model extractor for the formal description technique PROMELA, for the verification of C code using well defined APIs with the SPIN tool.
- The implementation of a compiler that generates an implicit labelled transition system for programs written in C. This transition system acts as an input formalism to the tools of the CADP verification enviroment [27].

Comparing our proposal with other work previously cited, BANDERA and JPF are oriented to the verification of JAVA programs. With regard to the tools for the C language, the most significant projects are: SLAM, BLAST, CMCy FEAVER. SLAM is a MODEL CHECKER specifically designed to verify that the behaviour of a *driver* (written in C) for the NT familly of operating systems is safe with respect to the use of the API offered by the operating system. CMC and FEAVER are oriented to C code verification with an event driven scheme, so they are suitables for verifying code for communications. The first one, CMC, is a MODEL CHECKER tool itself and it is not supported by any other external tool for the verification process. The

BLAST MODEL CHECKER follows, like SLAM, a counterexample-guided abstraction refinement (CEGAR) paradigm [16].

The most significant contribution of our methodology is that it is oriented to the verification of programs regarding the use of well defined APIs. Indenpendly of the API being considered, this method provides a guideline so that a model, described using some formal description technique and optimized both in the variables used and in the interleaving of statements of the processes, can verify the correct use of the external interface being considered. Specifically, following this methodology, we have implemented a model extractor for PROMELA to verify C code using the SPIN MODEL CHECKER. In addition, we have developed a compiler for translating C code into labelled transition systems to verify programs in the CADP enviroment.

With respect to the heap representation, the main contributions of the paper are:

- a novel method to deal with pointers and heap management in the context of C programs;
- the formalization of the method in order to prove its correctness;
- the implementation of our proposal in the SPIN model checker.

We propose a new representation of the heap of a given C process that consists of using an incremental global data structure to allocate new objects. This global store is not kept as a part of the state; instead, indexes to this store are only used to point to the store elements. The way we generate the indexes (a hashing method) and the way the store is managed allows us to efficiently manage canonical representations of the states and to implement model checkers with this feature. This approach constitutes a new way to implement state collapsing, and it does not affect the behavior of the model checkers when analyzing programs without pointers.

Previous works cited above only provided informal descriptions of the mechanism to manage dynamic memory. We describe our proposal giving formal semantics to the operations related to pointers and memory management. As far as we know, this is the first time that collapsing related methods has been formalized. As in [19], the formal semantics has been useful to check correctness and to guide the implementation.

Compared to related work, our method shares the ideas of the global store and canonical states with [37] and [11]. However, like JPF our way of managing the store is more efficient, because we save memory by keeping the real data outside the state vector. We also save time because: a) we do not need the linearization used in dSPIN and b) we do not need a specific mechanism to implement backtracking. This is due to the use of a special hashing method

to manage the global store. Furthermore, our method considers new features such as explicit memory deallocation and support for pointer arithmetic. Compared to BLAST our method considers the real contents of pointers and allows us to reason about the shape of the structures and their contents.

In connection with the verification of properties our main contributions are:

- A two dimensional logic that combines time (evolution of the program, e.g., by interleaving processes) and space (due to dynamic structures such as lists or trees).
- Specification of the graph types with CTL, characterizing the nodes in dynamic structures in such a way that propositions on pointers and propositions on data can easily be mixed in the same formula.
- An algorithm to perform on-the-fly model checking of MALTL formulas. The algorithm properly extends the standard algorithm to analyze LTL properties (described as Büchi automata) regarding the system behavior over time with a CTL algorithm to check properties about dynamic system data, exploring the graph of the dynamic structure.
- Implementation of the proposed logic as an extension to the SPIN model checker, in such a way that we can verify properties of non-trivial C programs.

Compared with other logic-based approaches, one difference in our proposal is the implementation with explicit on-the-fly model checkers. The methods in [11] and [34] are based on BDD verification. Other logics, like separation logic [47] or pointer assertion logic [43], have been implemented with theorem provers.

Our two proposals can be implemented in many existing tools. The implementation described in this thesis has been performed on SPIN and includes both the heap management and the model checking of MALTL formulas.

Conclusions and future work

Conclusions

In this thesis we have researched three relevant aspects in the verification of software using formal methods. First, we have developed a method for model extraction in programs that use of well defined interfaces. Secondly, we have dealt with the problem of verifying programs that use dynamic memory. In this area, we have worked in two differents and complementary ways: To provide a memory and pointer model to apply the MODEL CHECKING technique in programs using dynamic memory, and two-level logic that allows us to reason about shape and content in dynamic structures.

Regarding analysis of properties in programs that use well defined APIs, the methods presented in this thesis has facilitied the building of the SOCKETMC [19] tool to analyse programs that uses the well known SOCKETS interface for communication between processes. In addition, this technique has also allowed the verification with C.OPEN and ANNOTATOR [25] of applications written in C in the OPEN/CÆSAR environment. Moreover, this environment has been used within the FMICS-JETI [39, 40] platform for the integration and the use of remote tools.

As regards the verification of programs with dynamic memory, we have developed a model that manages the dynamic memory in two different areas: the *heap* that contains the logic information of the dynamic memory being used, and the *global store* that saves the data referenced in the heap. In this way, our method lets us to reduce the size of the state vector and also the spent time during the verification process, due to that the *matching* function responsible for the comparison between states does not need to check the global store, but only the heap. This approach also considers a pointer model suitable to verify C code, in order to properly manage the pointer arithmetic present in this language. Regarding verifying properties in dynamic structures, we have built the two level logic MALTL. This logic, inspired by the GRAPH TYPE [35]

formalism specifying properties in data structures, allows us to use the CTL logic to specify properties in dynamic structures. The analysis of these structures is done in time and space, in such a way that in each state during the verification of a temporal formulae, specified in the LTL logic, we also carry out the spacial verification of dynamic data structures. Thanks to the techniques presented in this work for the management of dynamic memory an the specification of properties in structures, we have performed the verification of the model of a driver written in C code for the LINUX operating system.

We should point out that from the very beginning we have sought to develop general approaches that could be independent of the implementation used an suitable for any tool. This has been shown in the implementation of the models extractor and the dynamic memory model. Both methods have been implemented in such different platforms as SPIN [33] and OPEN/CÆSAR [26] . The MALTL logic has only been implemented in SPIN, due to its integration with CADP [27], and because with this architecture, it would be necessary to modify the user module entrusted to do the verification in CADP (the tool EVALUATOR [41]) and it is not open source. Another option would be to build a specific user module to do the temporal analysis (the first logic level), and to integrate the second level in it.

Future work

Future work will focus on including new techniques to complete, and improve, the proposals presented in this thesis. On the one hand, the model extraction methodology for programs that use well defined interfaces need the formalization, as well as the model, of the well defined API being used. This is complex work and may take a considerable amount of development time. One way to simplify this process could be the use of a specification language ajusted for interface design (it must be understood within the context of programming language libraries). This will allow us to carefully and accurately describe the interface behaviour. The combination of a specification language with a tool that obtains an implentation from this language, would allow the interface models to be automatically obtained. Moreover, to make this task easier for the final user, it would be possible to provide a model library covering a wide number of interfaces.

Moreover, the dynamic memory model and the MALTL logic can be improved in several ways. The hashing method used in the dynamic memory management, although it reaches its

goal, it does not always get the a canonical heap representation. So, if two processes, concurrently, insert elements into a sorted list, the heap could be different depending on the statements interleaving. This means greater use of memory but the verification is still accurate. In addition, representing the heap is very expensive in memory usage. Although compression techniques are used in the state vector and indexes are used to represent the heap and the data, when we work with large structures, these techniques are not enough.

Despite of the compression techniques employed in the state vector, using indexes to represent the heap, and in the real data, using the global store, when we work with huge structures this techniques are not enough. Therefore, we plan as future work to research another hash function to avoid different representations for two configurations of one heap and to build an incremental representation of the heap to save memory usage in huge structures.

Finally, the CTL logic used in the second logic level in MALTL, could be weak when validating certains structures such as, for example, tree-like structures, the CTL logic may not be expressive enough to specify some properties which should be considered in certain elements of different branches. Although the CTL* logic can be more expressive than the CTL, its biggest advantage is that it offers the two levels, the spacial and the temporal, in one. However, as in our analysis we requiere the two levels to be separate, the CTL* logic is not suitable. Separation logic [47], on the other hand, adds a separation operator $P * Q$ that is satisfied if the propositions P and Q belong to disjoint portions (i.e , that they do not share data) in the storage structure. This is a basic approach to establish properties over general structures, achieving in this way both correctness and completeness over data structures. But the necessary modifications in the MALTL logic to introduce this operator are numerous because it is necessary to review the formal model of the dynamic memory and of the logic, as well as the implementation.

CONCLUSIONS AND FUTURE WORK

Bibliografía

[1] Rajeev Alur, Marcelo Arenas, Pablo Barcelo, Kousha Etessami, Neil Immerman, and Leonid Libkin. First-order and temporal logics for nested words. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.

[2] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, pages 467–481, 2004.

[3] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 332–341, New York, NY, USA, 2005. ACM.

[4] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[6] Michele Banci, Marcello Becucci, Alessandro Fantechi, and Emilio Spinicci. Validation coverage for a component-based sdl model of a railway signaling system. *Electr. Notes Theor. Comput. Sci.*, 116:99–111, 2005.

[7] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM.

[8] Brandon Bennett, Anthony G. Cohn, Frank Wolter, and Michael Zakharyaschev. Multi-dimensional modal logic as a framework for spatio-temporal reasoning. *Applied Intelligence*, 17(3):239–251, 2002.

[9] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007.

[10] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *CAV*, pages 221–225, 2007.

[11] A. Bouajjani, P Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Proc. of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, 2005.

[12] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Toms Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Static Analysis*, volume 2006, pages 52–70. Springer Verlag, 2006.

[13] Rémi Brochenin, Stéphane Demri, and Étienne Lozes. Reasoning about sequences of memory states. In *LFCS*, pages 100–114, 2007.

[14] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.

[15] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[16] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[17] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[18] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[19] P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined apis: the socket case. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 17–26, New York, NY, USA, 2005. ACM Press.

[20] Pedro de la Cámara, María del Mar Gallardo, and Pedro Merino. Model extraction for ARINC 653 based avionics software. In *SPIN*, pages 243–262, 2007.

[21] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.

[22] Dino Distefano, Peter W. Ohearn, and Hongseok Yang. A local shape analysis based on separation logic. In *In TACAS*, pages 287–302. Springer, 2006.

[23] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[24] Pascal Fradet and Daniel Le Métayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997. ACM.

[25] María-del-Mar Gallardo, Christophe Joubert, Pedro Merino, and David Sanán. C.open and annotator: Tools for on-the-fly model checking c programs. Tool session of the 14th International SPIN Workshop on Model Checking of Software SPIN'07 (Berlin, Germany), July 2007.

[26] H. Garavel. Open/caesar: An open software architecture for verification, simulation, and testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98,*, 1998.

[27] H. Garavel, F. Lang, and R. Mateescu. Cadp 2006: A toolbox for the construction and analysis of distributed processes. *In Proc. of CAV'07*, To appear.

[28] Garavel, H., Lang, F., and Mateescu, R. An overview of cadp 2001. In *EASST Newsletter*, number 4, pages 13–24, 2002.

[29] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[30] Laurie J. Hendren, Joseph Hummell, and Alexandru Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 249–260, New York, NY, USA, 1992. ACM.

[31] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN'2003), Portland, OR, USA, May 2003*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[32] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[33] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[34] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In *SPIN*, pages 299–305, 2006.

[35] Nils Klarlund and Michael I. Schwartzbach. Graph Types. In *POPL*, pages 196–205, 1993.

[36] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[37] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[38] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *In Proc. of VMCAI05, volume 3385 of LNCS*, pages 181–198. Springer, 2005.

[39] T. Margaria, R.Ñagel, and B. B. Steffen. Remote integration and coordination of verification tools in jeti. 2005.

[40] T. Margaria and B Steffen. Advances in the fmics-jeti platform for program verification. 2007. To appear in Proc. of ICECCS07.

[41] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.*, 46(3):255–281, 2003.

[42] Extracting Verification Models, Gerard J. Holzmann, Gerard J. Holzmann, Margaret H. Smith, and Margaret H. Smith. Software model checking. In *In Proceedings of FORTE/PSTV'99*, pages 481–497. Kluwer, 1999.

[43] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 221–231, New York, NY, USA, 2001. ACM Press.

[44] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[45] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[46] A. Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[47] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[48] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[49] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118, New York, NY, USA, 1999. ACM.

[50] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[51] Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In *In ESOP2003: European Symp. on Programming, volume 2618 of LNCS*, pages 204–222. Springer, 2003.