SPADE: Verification of Multithreaded Dynamic and Recursive Programs

Gael Patin, Mihaela Sighireanu, Tayssir Touili

SPADE: Verification of Multithreaded Dynamic and Recursive Programs [⋆]

Gaël Patin[1], Mihaela Sighireanu[2], and Tayssir Touili[2]

[1] University of Paris 7, Case 7014, 2 place Jussieu, 75251 Paris 05, France.
[2] LIAFA, CNRS & University of Paris 7, Case 7014, 2 place Jussieu, 75251 Paris 05, France.
{sighirea,touili}@liafa.jussieu.fr

1 Introduction

Recently, there are a lot of tools that have been considered for software verification. We can for example mention BLAST [HJMS02], SLAM [BR01], KISS [QW04,QR05], ZING [QRR04], and MAGIC [CCG$^+$03,CCG$^+$04,CCK$^+$06]. However, none of these tools can deal with parallelism, communication between parallel processes, dynamic process creation, and recursion at the same time. The tool we propose, called SPADE, allows to analyse automatically boolean programs presenting all these features. As far as we know, this is the first software model checking tool based on an expressive model that *accurately* models all these aspects in programs.

SPADE checks safety properties of programs by iteratively refining abstractions of the sets of the program execution paths that violate the property. Since property checking is undecidable for programs presenting all the features mentioned above, the SPADE refinement algorithm may not converge. In case of convergence, it can either find a bug in the program and returns a counterexample to the user, or certify that the program is correct.

We have applied SPADE to different case studies. Our results are encouraging and are reported in Section 4. In particular, we were able to *automatically* find two bugs in two versions of a Windows NT Bluetooth driver. The bugs were already found in [CCK$^+$06]. But there, the verification was not *completely* automatic since the authors needed to *guess* the number of processes for which the bugs occur. Whereas with SPADE, the verification process was done in a *completely* automatic manner. Indeed, we don't need to make any guess since our tool handles *dynamic creation of processes*.

The current version of SPADE is available at http://www.liafa.jussieu.fr/∼sighirea/spade.

2 The underlying techniques

SPADE is based on the SPAD model [Tou05]. A SPAD is a finite set of rules of the form $t \xrightarrow{a} t'$, where a is a synchronisation action, t and t' are terms built up from the null process "0", a finite number of variables (X), the sequential composition "$\cdot$", and

[⋆] This work has been supported by the French Governement program ACI Jeunes Chercheurs, Contract No.02 2 0205.

the asynchroneous parallel composition "$||$", where the operators "$\cdot$" and "$||$" are respectively associative and associative/commutative, and where each action a has its corresponding co-action $\bar{a}$. Intuitively, the process "0" represents termination, a process variable X corresponds to a control point of the program, and a process term t describes the control structure of the program. A procedure call is represented by a rule of the form $X \to Y \cdot Z$, where the program at control point X calls the procedure Y and goes to control point Z. This control point Z becomes active when Y terminates. Dynamic creation of parallel processes is modeled by rules of the form $X \to Y||Z$, expressing that a process in control point X can create two parallel processes in control points Y and Z, respectively. Finally, handshakes between parallel processes are represented according to the CCS style by rules of the form $t_1 \xrightarrow{a} t_1'$ and $t_2 \xrightarrow{\bar{a}} t_2'$, meaning that two parallel processes t_1 and t_2 can synchronize and move simultaneously to t_1' and t_2', respectively.

SPADE deals with rechability queries for SPAD models. More precisely, given two (possibly infinite) sets of configurations *Init* and *Bad*, the problem is to know whether the set of bad configurations *Bad* can be reached from the initial configurations *Init*. The approach implemented in SPADE consists in computing abstractions of the execution path language that leads form *Init* to *Bad* and iteratively refining these abstractions [Tou05]. Our techniques are based on (1) the representation of the sets of configurations with binary tree automata, (2) the use of these automata to compute a set of constraints whose least fixpoint characterize the set of execution paths of the program, and (3) the resolution of this set of constraints in an abstract domain. Our algorithm is generic and can deal with different abstract domains. In particular, we considered the domains D_n of finite action words of length less or equal to n. These domains allow to compute abstractions of the execution paths that are exact up to the depth n. These abstractions are called n-prefix abstractions. The refinement step consists in considering a "more precise" abstract domain by incrementing the depth n.

3 The SPADE tool

SPADE has two inputs. The first input is an ASCII file describing (1) the SPAD model of the program (names of processes, names of actions, rewriting rules), (2) the (possibly infinite) set of initial configurations *Init* (given by a tree automaton), and (3) the bad configuration *Bad* (a tree automaton). The second input is optional and consists of an integer that represents the depth n of the prefix abstraction. If this parameter is not given by the user, the tool starts with a prefix abstration of depth one, and automatically increases the abstraction depth until either an error is found or the program is proven to be correct.

SPADE outputs (a) the language $reach_n$ representing the n-prefix abstraction of the paths between *Init* and *Bad*, and (b) the result of the intersection of $reach_n$ with the set of *good* execution paths. This result may be either (CANNOT) if the intersection is empty (i.e., the n-prefix abstraction does not allow to find an execution leading from *Init* to *Bad*), (MAYBE) if the intersection is not empty but the path found has been cut by the abstraction, (CAN) if a *real* path (i.e., not cut by abstraction) has been found between *Init* and *Bad*.

SPADE implements in OCAML the algorithm described in [Tou05]. OCAML provides a rich and efficient built-in library of data structures (e.g., hash tables, maps, sets), a powerful system of modules, and garbage collection facilities. Due to these features, the algorithm is implemented as a *generic* module parameterized by two *signatures* (interfaces): the first signature collects types and operations dealing with tree automata, and the second signature collects types and operations of the abstract domain of execution paths. The current version of SPADE instantiates the first parameter of the algorithm with the OCAML implementation of tree automata provided by the TIMBUK tool [GT01]. This implementation provides a large list of operations on tree automata (union, intersection, emptiness test, minimization, etc) and an easy access to the states and the transitions of automata. For the second parameter, we implemented in OCAML a library for the abstract domain D_n (i.e., finite sets of finite words of length less or equal to n). The library provides efficient implementation of operations intensively used by the algorithm: union, concatenation, shuffle, prefix, and inclusion.

4 Summary of the results

SPADE has been applied to several examples. The performances are given in Table 1. The experiments were obtained on a 4GHz Pentium IV with 4GB of memory.

Example	Time	Space
BlueTooth v1	1623mn28s	50 MB
BlueTooth v2	1216mn28s	46 MB
ConcVector v1	7s	3.4 MB
ConcVector v2	14s	14.8 MB
Lock/unlock	8s	3.6MB

Table 1. Performances of SPADE

The *BlueTooth* v1 is the SPAD model of the BlueTooth driver program used by Windows NT and given in [QW04]. We were able to find a bug in this program. To find this error, the [QW04] authors needed to guess the number of driver's requests for which the error occurs, and then run their tool; whereas with SPADE, the verification was done in a *completely* automatic manner, since we did not have to guess the number of requests for which the error occurs because our tool can deal with *dynamic creation of processes*.

The *BlueTooth* v2 is a corrected version of *BlueTooth* v1 proposed by the authors of [QW04]. SPADE finds an error in this version as well. This bug was already found in [CCK$^+$06]. Again, to be able to find the bug, the authors of [CCK$^+$06] needed to guess the number of requests that causes the bug before running their tool, whereas SPADE did not need to perform this guess.

ConcVector is a SPAD model of a multithreaded program using concurrently methods of the class `java.util.Vector` from the Java Standard Collection Framework.

The program's threads create and remove the elements of a `Vector` object. Wand and Stoller [WS03] reported a high-level data race that occurs on such programs because the constructor of the `Vector` class is not atomic. SPADE found this bug for a program with an unbounded number of threads (ConcVector v1). Version v2 fixes the bug by taking an atomic implementation of the constructor. SPADE was able to prove that this version is correct.

The *Lock/unlock* example is a system that handles an *arbitrary* number of concurrent insertions on a binary search tree. The algorithm was proposed in [KL80], and can be applied to handle simultaneous insertions (done by several users) into a database, or to reduce the time necessary for a single insertion. We considered a buggy version of the algorithm where one or several processes do not adhere to the required lock and unlock policy. This version was considered in [CCK$^+$06], where the bug was found *only* for systems where the number of concurrent processes is less or equal to 7. With SPADE, we were able to check this buggy program for *arbitrary* number of concurrent insertion processes.

References

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.

[CCG$^+$03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[CCG$^+$04] Sagar Chaki, Edmund Clarke, Orna Grumberg, Joel Ouaknine, Natasha Sharygina, Tayssir Touili, and Helmut Veith. An expressive framework for state/event systems. Technical report, Carnegie Mellon University, 2004.

[CCK$^+$06] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 2006.

[GT01] Thomas Genet and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 2001.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[KL80] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *proceedings of TACAS'05*, 2005.

[QRR04] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245–255, 2004.

[QW04] S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24, 2004.

[Tou05] T. Touili. Dealing with communication for dynamic multithreaded recursive programs. In *1st VISSAS workshop*, 2005. Invited Paper.

[WS03] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2003.