Introduction to StEAM - An Assembly-Level Software Model Checker

Tilman Mehler[1] and Peter Leven[2]

[1] Universität Dortmund, Baroper Str. 301, 44227 Dortmund, Germany
tilman.mehler@cs.uni-dortmund.de
[2] Institut für Informatik, Georges-Köhler Allee, Geb. 51,
Albert-Ludwigs-Universität Freiburg, Germany
leven@informatik.uni-freiburg.de

Abstract. Model checking techniques have been used successfully for the exploration of state-based systems to search systematically for errors. As a recent trend, model checking was applied for the verification of software. Most of the current approaches are based on the (semi-)automatic generation of an abstract model from the source code. Such an abstraction may not correctly reflect the behavior of the actual program. This means, the abstract model may contain errors which are not present in the actual program and vice versa. Some newer approaches - such as the Java model checker JPF - avoid this problem, by checking the actual program instead of an abstract model. In our own work we focus on the mainly used programming language C++. The usage of a virtual machine for C++ and our extension to multi threading allows platform independent model checking of thread based C++-programs. First examples show, that our system *StEAM* [3] can successfully be used to detect deadlocks and assertion violations.

1 Introduction

Model checking [4] describes the process of generating all possible configurations of a system with the intention to prove, that it satisfies one or more formal properties. After successful application in fields like hardware design, process engineering and protocol verification some recent efforts [16,11,18] exploit model checking techniques for the verification of actual programming languages like Java or C++. Many of these approaches are based on the extraction of a formal model from the source code of the program. Such a model can in turn be converted into the input language of an existing model checker (e.g. Spin [15]).

A main drawback of these model checkers lies in the fact, that most of them cannot handle the dynamic aspects of actual computer programs, like

[3] State Exploring Assembly Model Checker

memory allocation and dynamic object instantiation. Therefore, a direct correspondence between the actual program and the generated model cannot always be guaranteed. Also, some of these approaches require manual intervention of the user, which in turn can mean that he/she must be familiar with the input language of the underlying model checker. A different approach is made by the Java model checker JPF[4]. Instead of extracting a model from the source code, JPF [18] uses a custom-made Java virtual machine (JVM) to check a program on the Bytecode level. This eliminates the problem, that there is possibly no direct correspondence between the checked model and the actual program. StEAM, the tool presented in this paper makes a more general approach to such low-level program model checking. Based on a virtual processor, called *Internet Virtual Machine* (IVM), the tool is capable to perform a search on machine-code compiled from a C/C++ source. One the one hand, this provides the possibility to model check programs written in the industrial standard programming language, on the other hand, the generic approach of StEAM is extendible to any compiler-based programming language with reasonable effort.

The paper is structured as follows. First, it introduces the Internet Virtual Machine. Next, it shows, which extensions to IVM have been made to perform model checking, also by using illustrating examples of error detection. The state exploration and the state representation of *StEAM* is explained detailed. Experiments show, that *StEAM* can be used successfully for model checking C++-programs and indicated future improvements. *StEAM* is related to other work in the area of model checking and finally we summarize and draw conclusions.

2 ICVM

The ICVM project by Bob Daley came to life with the wish to create a programming language that provides platform-independency without the need of rewriting applications into proprietary languages like $C\sharp$ or Java. The main purpose of ICVM was to be able to receive precompiled programs through the Internet and run them on an arbitrary platform without recompilation. Furthermore ICVM was mainly designed to run games, so speed was an important issue.
The two main components of ICVM are the compiler and the virtual machine (VM), called IVM. The compiler takes conventional C/C++ code and translates it into the machine code of the VM. The VM simulates a 32-bit CISC CPU with a set of approximately 64,000 instructions. The current version of ICVM is already capable of running complex programs at descend speed, including the commercial game Doom[5]. ICVM is publicly available as open source[6].

[4] NASA AMES Research Center
[5] http://www.doomworld.com/classicdoom/ports/
[6] http://ivm.sourceforge.net/

2.1 Structure and State of IVM

ICVM uses a modified version of the GNU C-compiler gcc to compile its programs. The resulting code is stored in ELF-format 2.1. For a clear understanding of the working method and state of IVM, a short introduction to this format is necessary.

The ELF Format ELF[7] (Executable and Linking Format) represent the common object file format for Linux binaries. The three types of files representable are .o-files, shared libraries and executables. We will concentrate on files of the latter type here. An ELF-binary is partitioned in so-called 'sections' describing different aspects of the object's properties. The number of sections vary depending on the respective file. For us, the two most important sections are the *.data*- and *.bss*-section (in the following: DATA/BSS). Together, the two sections represent the set of global variables of the program. The BSS-section describes the set of un-initialized variables while the DATA-section stands for the set of variables that have an initial value assigned to them. When the program is executed, the system first loads the ELF file into memory. For the BSS-section additional memory must be allocated, since the un-initialized variables do not occupy space in the ELF file itself. In contrast, the space for initialized variables is reserved in the DATA-section of the object file itself, so accesses to the variables directly affect the memory image of the ELF binary. Other sections represent the executable code, the symbol table etc. and need not to be discussed here.

State of a Program Execution in IVM. Figure 1 illustrates the components that form a state during the execution of a program compiled for IVM. The system's memory is divided in two hierarchic subsets - one describing the memory regions visible to the program executed by the VM and the set which is visible only for the VM itself. In the case of the unmodified IVM, the latter set corresponds to the physical memory. The *machine* corresponds to the CPU in actual computer architectures. From the view of the physical machine - i.e. the computer that executes the virtual machine - this is simply an instance of the structure *ICVMMachine*, which is used to store the contents of several machine registers. The machine registers include the program counter (PC), the stack pointer (SP), the frame pointer (FP), as well as some general purpose registers used for integer and floating-point arithmetic. The PC indicates the byte offset of the next machine instruction to be executed, i.e. it refers to some position within the code-section of the object file's memory image (MI). The SP and FP refer to a position of the stack, which is a linear region in physical memory. The stack is used for storing the return addresses of function calls and the contents of local variables. Furthermore the MI contains the DATA- and BSS-sections. Note, that (in contrast to the DATA-section) the space for storing the contents of the

[7] http://www.shrubbery.net/solaris9ab/SUNWdev/LLM/p36.html

variables declared in the BSS-section lies outside the MI and is allocated separately. The circles inside program memory denote the memory regions that have been allocated during program execution (e.g. by *malloc(..)*) and not deallocated up to the current state. These also include class instances created by a *new* constructor. Given the above, a state during the execu-

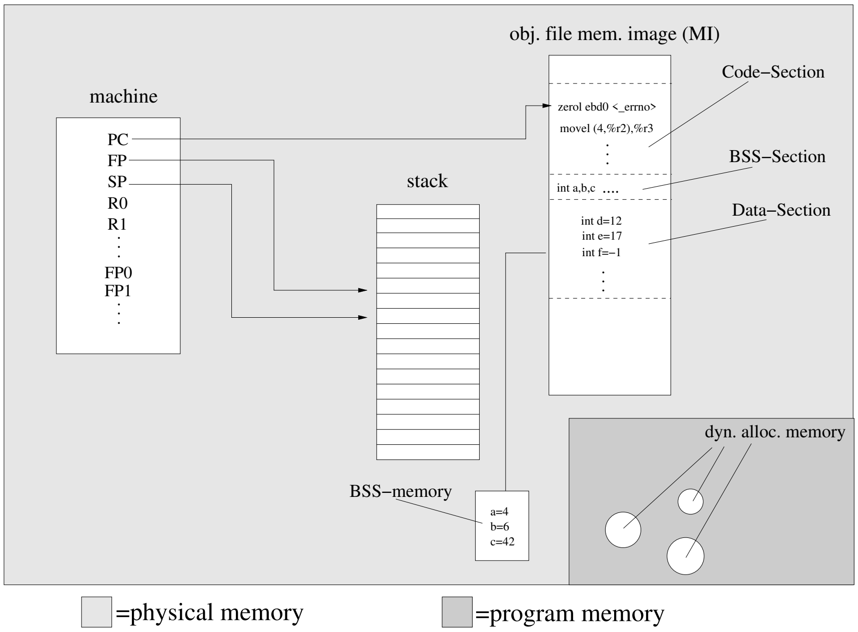

Fig. 1. System state from the view of IVM.

tion of an IVM-program can be described as (m, s, BS, DS, DM), where m denotes the contents of the machine registers, s the content of the stack, BS and DS the contents of the global variables (defined in the BSS- and DATA-section) and DM stands for the dynamically allocated memory regions. Note that this description is only implicit in the implementation of IVM.

3 From IVM to StEAM

This section illustrates the extensions necessary to turn IVM into a model checker. As in Section 2 we visualize the components of the enhanced system and give a formal description of the system state. Afterwards we use some simple example programs to give a demonstration of the capabilities of StEAM.

3.1 Extensions to IVM

In the course of this project, IVM was extended with a multi-threading capability, a description of the search space of a program, as well as some special-purpose program statements, which enable the user to describe and guide the search. Figure 2 shows the components of the enhanced system. As indicated, an additional control/memory layer is introduced. The memory

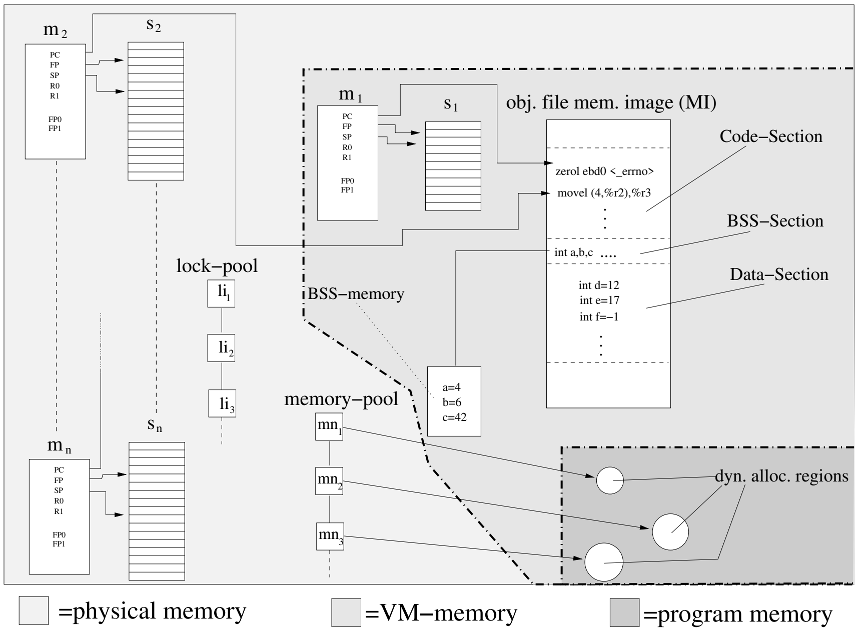

Fig. 2. System state form the view of *StEAM* .

region which in Figure 1 represented the physical memory, is now denoted as VM-memory and the new physical region is visible only by *StEAM* itself. Due to the new multi-threading capabilities we now have n stacks $s_1, \ldots, s_n$ and n machines $m_1, \ldots, m_n$ (one for each running thread). The structures s_1 and m_1 correspond to the main thread and are created by IVM when the verification process starts. Therefore they reside in VM-memory. From the running threads, new threads can be dynamically created. Such a creation is recognized by the control layer of *StEAM* through a specific pattern of machine instructions. Furthermore, we have two new structures, namely the memory- and lock-pool. The memory pool is used by *StEAM* to manage the dynamically allocated memory. It consists of a list of entries (memory nodes) $mn_1, mn_2, \ldots$ (one for each memory region) which contain a pointer to the region itself as well as some additional information (such as the the identity of the thread, from which it was allocated). The lock-pool stores information about locked resources (an example will follow in Section 3.3). As soon as more than one thread is running, *StEAM* is responsible for switch-

ing control between them. This means, before the next step of a thread can be executed, the content of the machine registers and the stack must be set to the state immediately after the last execution of the same thread - or directly after the thread was created if it is new. It must be clearly pointed out, that there is a significant difference between the simple execution of a multi threaded program and the exploration of its state space. In the first case, it suffices to restore the machine registers and the stack content of the executed thread. In the latter case, *StEAM* must also restore the state of the Data- and BSS-sections, as well as the memory and lock pool. Although *StEAM* will also support program execution, in the following we are only interested in exploration. Combined, this leads to a state description $(m_1, \ldots, m_n, s_1, \ldots, s_n, BS, DS, DM, LP)$, where m_i and s_i describe the stack contents of a thread, the components BS, DS, DM have the same meaning as in explained in section 2 and LP stands for the lock pool.
On the programming level, multi threading is realized by the creation of a base class *ICVMThread*, from which all thread classes are to be derived. A class derived from ICVMThread must implement the methods *start()*, *run()* and *die()*. After creating an instance of the derived thread-class, a call to *start* will initiate the thread execution. The run method is called from within the *start*-method and must contain the actual loop of the thread.

3.2 Detection of Assertion Violations

Figure 3 shows a simple program *glob* which generates two threads from a derived thread class *MyThread*, that access a shared variable *glob*. The main program uses an atomic block of code to create the threads. Such a block is defined by a pair of BEGINATOMIC and ENDATOMIC statements. Upon creation, each thread is assigned a unique identifier *ID* by the constructor of the super class. An instance of *MyThread* uses *ID* to apply the statement '`glob=(glob+1)*ID`' once. The *main*-method contains a VASSERT-statement. This statement takes a boolean expression as its parameter and acts like an assertion in established model checkers like e.g. SPIN [15]. If *StEAM* finds a sequence of program instructions (the *trail*) which leads to the line of the VASSERT-statement, and the corresponding system state violates the boolean expression, the model checker prints the trail and terminates.

In the *glob*-example we check against the expression `glob!=8`. Figure 4 shows the output of *StEAM* , when applied to *glob*.

Thread 1 denotes the main thread, Thread 2 and 3 are the two instances of *MyThread*. The returned error trail is easy to trace. First, the instances of *MyThread* are generated and started in one atomic step. Then the one-line *run*-method of Thread 3 is executed, followed by the *run*-method of Thread 2. We can easily calculate, why the assertion is violated. After Step 3, we have $glob := (0 + 1) * 3 = 3$ and after step 5 $glob = (3 + 1) * 2 = 8$. After this, the line containing the VASSERT-statement is reached. Obviously the assertion is only violated, if the *main*-method of Thread 3 is executed before that of Thread 2. Otherwise *glob* would take the values $0, 2, 9$.

```
#include "IVMThread.h"
#include "MyThread.h"
extern int running[99];
extern int glob;
class IVMThread;
MyThread::MyThread() : IVMThread::IVMThread() {
}
void MyThread::start() {
 run();
 die();
}
void MyThread::run() {
  glob=(glob+1)*ID;
}
void MyThread::die() {
}
int MyThread::id_counter;
```

```
    .
 (comments)
    .
    .
#include <assert.h>
#include "MyThread.h"
#define N 2
class MyThread;
MyThread * t[N];
int glob=0;
int i;
void initThreads () {
 BEGINATOMIC
 for(i=0;i<N;i++) {
   t[i]=new MyThread();
   t[i]->start();
  }
  ENDATOMIC
}
void main()
{
 initThreads();
 VASSERT(glob!=8);
}
```

Fig. 3. The source of the program 'glob'.

```
Data - Section - size: 520
BSS  - Section - size: 2416
Using Depth-First search!
Adding thread No. 1 at PC: 5d4
Termination point at PC: 656
Adding thread No. 2 at PC: 26e
Termination point at PC: 2f0
Adding thread No. 3 at PC: 26e
Termination point at PC: 2f0
Assertion violation, printing trail!
depth: 9
Step 1: Thread 1 - Line 29 source-file: glob.c -  initThreads
Step 2: Thread 1 - Line 36 source-file: glob.c -  main
Step 3: Thread 3 - Line 21 source-file: MyThread.cc -  MyThread::run
Step 4: Thread 3 - Line 22 source-file: MyThread.cc -  MyThread::run
Step 5: Thread 2 - Line 21 source-file: MyThread.cc -  MyThread::run
Step 6: Thread 2 - Line 22 source-file: MyThread.cc -  MyThread::run
Step 7: Thread 1 - Line 41 source-file: glob.c -  main
Step 9: Thread 1 - Line <unknown> source-file: glob.c -  main
************************************************
*                                              *
*           STATISTICS                         *
*                                              *
************************************************
Searchalgorithm : DFS

States visited        : 10

Memory usage  (byte)  : 100737564
Memory usage  (kB)    : 98376
Memory usage  (M)     : 96

Number of hash-codes  : 19

Usage of  most frequent hash-code: 1
Usage of least frequent hash-code: 1
 Computation Time in msecs : 450
  Time Difference in secs : 0.00
```

Fig. 4. The output for the 'glob'-program.

3.3 Detecting Deadlocks

The model checker *StEAM* automatically checks for deadlocks during the program exploration. A thread can gain and release exclusive access to a resource using the statements VLOCK and VUNLOCK, which take as their parameter a pointer to a base type or structure. When a thread attempts to lock an already locked resource, it must wait until the lock is released by the thread which holds it. A deadlock in *StEAM* describes a state, where all running threads wait for a lock to be released. We will illustrate this by another simple example, namely the well-known dining philosophers problem [10,7]. This scalable problem describes n philosophers sitting at a round table. Each philosopher has a plate of spaghetti in front of him and there is one fork lying between each pair of philosophers, meaning n forks altogether. Each philosopher takes turns of meditation and eating phases. At the end of a meditation phase, a philosopher tries to take the fork to his left. If the fork is already taken, he waits until it becomes available again. After this, he tries to take the fork to his right and waits again, if necessary. When both forks were successfully taken, the philosopher starts eating. After he finished eating, he drops both forks and starts a new meditation phase. Figure 5 shows the source code of our C++-implementation of the dining philosophers. In this implementation, we use global *short int* variables for the forks, but will do any arbitrary base type or structure. As we will see, a deadlock exists in the state space of this program. Figure 6 shows the output of *StEAM* when applied to the philosophers program. Although the checked program is very simple, the produced trail is long (45 steps) and rather incomprehensible. The reason for this, lies in the fact, that by default, *StEAM* uses depth first search (DFS) for the exploration of a program. In general, DFS finds an error quickly while having low memory requirements. As a drawback, the error trails found with DFS can become very long, in some cases even too long to be traceable by the user. In the current version, *StEAM* supports DFS and breadth-first search (BFS). Figure 7 shows the output of the model checker when BFS is used.

The trail produced by using BFS is now only 16 steps long, but more states where visited and more memory was required than with DFS. The trail from Figure 7 is now easy to trace. After Thread 1 has started Thread 2 and 3, Thread 2 (Philosopher#1) executes up to line 27 (i.e. until the first fork was picked up). Immediately after this, the same happens for Philosopher#2. After this, the main thread terminates and the two philosophers try to take the respective other fork and the deadlock occurs. A minor modification of the source code makes the program deadlock free; the two VLOCK instructions in *Philosopher.cc* are included into one atomic block, i.e.:

```
BEGINATOMIC
VLOCK(leftfork);
VLOCK(rightfork);
ENDATOMIC
```

```
#include "icvm_verify.h"
#include "IVMThread.h"
#include "Philosopher.h"


class IVMThread;
extern int g[10];

Philosopher::Philosopher(short * lf, short * rf) : IVMThread::IVMThread() {

  leftfork=lf;
  rightfork=rf;

}

void Philosopher::start() {
  run();
}

void Philosopher::run() {

  while(1) {

    VLOCK(leftfork);
    VLOCK(rightfork);
    VUNLOCK(rightfork);
    VUNLOCK(leftfork);
  }
}

void Philosopher::die() {}

int Philosopher::id_counter;
```

```
        .
     (comments)
        .
#include <stdlib.h>
#include <assert.h>
#include "Philosopher.h"

class Philosopher;

Philosopher ** p;
short * forks;

void initThreads (int n) {
  p=(Philosopher **) malloc(n*sizeof(Philosopher *));
  forks=(short *) malloc(n*sizeof(short));
  for(int i=0;i<n;i++) {
    p[i]=new Philosopher(&forks[i], &forks[(i+1) % n]);
    p[i]->start();
  }
}

void main(int argc, char ** argv)
{
  int n;
  BEGINATOMIC;
  if(argc<2) {
    fprintf(stdout, "---=====Missing Parameter!=====---\n");
    exit(0);
  }
  n=atoi(argv[1]);
  initThreads(n);
  ENDATOMIC;
}
```

Fig. 5. The Source for the dining philosophers.

```
Data - Section - size: 512
BSS  - Section - size: 2424
Using Depth-First search!
Adding thread No. 1 at PC: 6fc
Termination point at PC: 86e
Adding thread No. 2 at PC: 2b0
Termination point at PC: 476
Adding thread No. 3 at PC: 2b0
Termination point at PC: 476
Deadlock detected, printing trail!
depth: 45
Step 1: Thread 1 - Line 34 source-file: philosophers.c -  main
Step 2: Thread 1 - Line 42 source-file: philosophers.c -  main
Step 3: Thread 3 - Line 23 source-file: Philosopher.cc -  Philosopher::run
Step 4: Thread 3 - Line 24 source-file: Philosopher.cc -  Philosopher::run
Step 5: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 6: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 7: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 8: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 9: Thread 3 - Line 29 source-file: Philosopher.cc -  Philosopher::run
Step 10: Thread 3 - Line 30 source-file: Philosopher.cc -  Philosopher::run
Step 11: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 12: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 13: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 14: Thread 2 - Line 23 source-file: Philosopher.cc -  Philosopher::run
Step 15: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 16: Thread 3 - Line 29 source-file: Philosopher.cc -  Philosopher::run
Step 17: Thread 3 - Line 30 source-file: Philosopher.cc -  Philosopher::run
Step 18: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 19: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 20: Thread 2 - Line 24 source-file: Philosopher.cc -  Philosopher::run
Step 21: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 22: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 23: Thread 3 - Line 29 source-file: Philosopher.cc -  Philosopher::run
Step 24: Thread 3 - Line 30 source-file: Philosopher.cc -  Philosopher::run
Step 25: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 26: Thread 2 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 27: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 28: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 29: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 30: Thread 3 - Line 29 source-file: Philosopher.cc -  Philosopher::run
Step 31: Thread 3 - Line 30 source-file: Philosopher.cc -  Philosopher::run
Step 32: Thread 2 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 33: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 34: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 35: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 36: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 37: Thread 3 - Line 29 source-file: Philosopher.cc -  Philosopher::run
Step 38: Thread 2 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 39: Thread 3 - Line 30 source-file: Philosopher.cc -  Philosopher::run
Step 40: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 41: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 42: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 43: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 44: Thread 2 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 45: Thread 1 - Line 42 source-file: philosophers.c -  main
************************************************
*                                              *
*           STATISTICS                         *
*                                              *
************************************************
Searchalgorithm : DFS

States visited        : 50

Memory usage  (byte)  : 101010416
Memory usage  (kB)    : 98642
Memory usage  (M)     : 96

Number of hash-codes  : 122

Usage of  most frequent hash-code: 2
Usage of least frequent hash-code: 1
 Computation Time in msecs : 460
  Time Difference in secs : 0.00
```

Fig. 6. The output for the dining philosophers.

```
Data - Section - size: 512
BSS  - Section - size: 2424
Using Breadth-First search!
Adding thread No. 1 at PC: 6fc
Termination point at PC: 86e
Adding thread No. 2 at PC: 2b0
Termination point at PC: 476
Adding thread No. 3 at PC: 2b0
Termination point at PC: 476
Deadlock detected, printing trail!
depth: 14
Step 1: Thread 1 - Line 34 source-file: philosophers.c -  main
Step 2: Thread 1 - Line 42 source-file: philosophers.c -  main
Step 3: Thread 2 - Line 23 source-file: Philosopher.cc -  Philosopher::run
Step 4: Thread 2 - Line 24 source-file: Philosopher.cc -  Philosopher::run
Step 5: Thread 2 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 6: Thread 2 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 7: Thread 2 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 8: Thread 3 - Line 23 source-file: Philosopher.cc -  Philosopher::run
Step 9: Thread 3 - Line 24 source-file: Philosopher.cc -  Philosopher::run
Step 10: Thread 3 - Line 25 source-file: Philosopher.cc -  Philosopher::run
Step 11: Thread 3 - Line 26 source-file: Philosopher.cc -  Philosopher::run
Step 12: Thread 3 - Line 27 source-file: Philosopher.cc -  Philosopher::run
Step 13: Thread 2 - Line 28 source-file: Philosopher.cc -  Philosopher::run
Step 14: Thread 3 - Line 28 source-file: Philosopher.cc -  Philosopher::run
************************************************
*                                              *
*           STATISTICS                         *
*                                              *
************************************************
Searchalgorithm : BFS

States visited       : 265

Memory usage  (byte)  : 102206592
Memory usage  (kB)    : 99811
Memory usage  (M)     : 97

Number of hash-codes  : 263

Usage of  most frequent hash-code: 3
Usage of least frequent hash-code: 1
 Computation Time in msecs : 530
  Time Difference in secs : 0.00
```

Fig. 7. BFS output for the dining philosophers.

Figure 8 shows the output of *StEAM* for the corrected dining philosophers program.

```
Data - Section - size: 512
BSS  - Section - size: 2424
Using Breadth-First search!
Adding thread No. 1 at PC: 744
Termination point at PC: 8b6
Adding thread No. 2 at PC: 2b0
Termination point at PC: 4be
Adding thread No. 3 at PC: 2b0
Termination point at PC: 4be
************************************************
*                                              *
*           STATISTICS                         *
*                                              *
************************************************
Searchalgorithm : BFS

States visited        : 199

Memory usage  (byte)  : 101885900
Memory usage  (kB)    : 99497
Memory usage  (M)     : 97

Number of hash-codes  : 161

Usage of  most frequent hash-code: 4
Usage of least frequent hash-code: 1
 Computation Time in msecs : 510
  Time Difference in secs : 0.00
Verified!
```

Fig. 8. Output for the corrected dining philosophers.

4 State Space of a Program

In the preceding sections, we gave an introduction to the structure and features of *StEAM* supported by some straightforward example programs. In the following, we discuss, how *StEAM* represents and explores its search space.

4.1 State Representation

The class diagram in Figure 9 reflects the state components from Figure 2 as represented by *StEAM* . Such a state is essentially composed of a set of *thread states*, two *stored sections* dsec/bsec as well as two pools containing the dynamically allocated memory and the set of locked resources. Each thread state contains a *stored stack* and an instance of *ICVMMachine*. A structure *stored stack* is used to memorize the current content of a thread's stack, ICVMMachine stores the values of the machine registers. The stored sections contain the initialized and un-initialized global variables (DATA- and BSS-section). For the sake of memory-efficiency, distinct system states can have common thread states or stored sections and two different thread states can share the same stored stack (hence the *user*-counter). For improved efficiency, the lock- and memory-pool of a system state are implemented as AVL-trees [1].

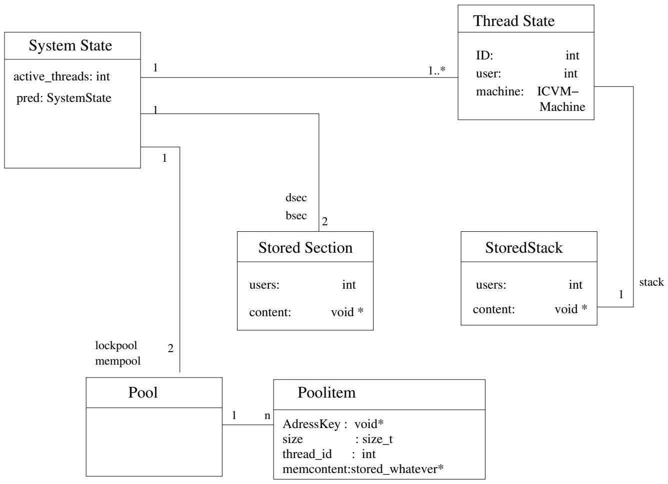

Fig. 9. State representation of *StEAM* .

Figure 10 shows a search tree up to Level 2, as produced for the *glob* example. Here, the components of each state are depicted as symbols of different shape. Ovals symbolize a whole state. The Data-section is depicted as a diamond, circles denote BSS-sections. A thread state is symbolized by a square, a stack by a semi-circle and a cross denotes the memory pool. We will disregard the lock pool here, because no locks occur. A dot in place of a symbol means, that the corresponding state component is the same as in the predecessor state. The initial state S_0 corresponds to the program state with only the main thread running. Apparently, S_0 has only one successor state - $S_{1,1}$ - which results from iterating the main thread. As shown in Figure 3, this calls and executes the *initThreads* method in one atomic step. As the result $S_{1,1}$ has three running threads. Only the Data-Section is modified, because the main thread writes to the un-initialized class pointers $t[i]$. The BSS-section remains unchanged. The stack of the main thread is changed by using the local index variable i. The memory pool is changed, because the call of *new*-constructor during thread generation allocates memory for the instances of the *MyThread*-class. From $S_{1,1}$ we have three potential successor states, $S_{2,1}$ $S_{2,2}$ and $S_{2,3}$ that correspond to executing one of the running threads. Iterating the main thread executes the VASSERT statement leading to state $S_{2,1}$. This will only affect the program counter of the main thread and thus change the main-thread's state. All other components, including the main thread's stack, remain unchanged. The states $S_{2,2}$ and $S_{2,3}$ correspond to iterating one of the dynamically generated threads. This executes the statement '`glob=(glob+1)*ID`', alternating the zero-initialized global variable glob, and thus the BSS-Section. All other components - with the exception of the respective thread state - remain unchanged. During exploration, *StEAM* uses a hash table to store already visited states. When expanding a state, only those successor states not in the hash table are added to the search tree. If the expansion of a state S yields no new states, then S forms a leaf in the search tree.

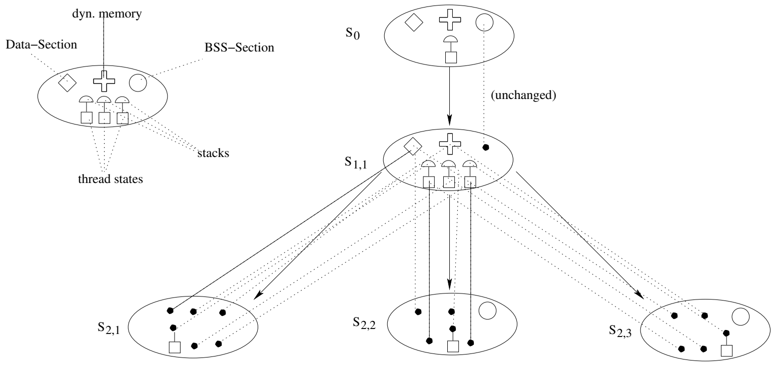

Fig. 10. The search tree of the glob example.

5 Summary of StEAM Features

In this section, we will summarize the features of *StEAM* .

5.1 Expressiveness

Due to the underlying concept which uses a virtual machine to interpret machine code compiled by a real C/C++ compiler[8] there are in principle no syntactic or semantic restrictions to the programs that can be checked by *StEAM* , as long as they are valid C/C++ programs.

[8] IVM uses a modified version of GCC to compile its code.

5.2 Multi-Threading

As opposed to Java, there is currently no real standard for multi threading in C++. We decided to implement a simple multi threading capability our own. To create a new Thread class, the user must derive it from the base class *ICVMThread* and implement the methods *start()*, *run()* and *die()*, as it was done in e.g. Figure 3. If in the near future a standard for multi threading will arise, *StEAM* will support it.

5.3 Special-Purpose Statements of StEAM

The special-purpose statements in *StEAM* are used to define properties and to guide the search. In general, a special-purpose statements is allowed at any place in the code, where a normal C/C++ statement would be valid[9].

BEGINATOMIC
This statement marks an atomic region. When such a statement is reached, the execution of the current thread will be continued until a consecutive ENDATOMIC. A BEGINATOMIC statement within an atomic block has no effect.

ENDATOMIC
This statement marks the end of an atomic region. An ENDATOMIC outside an atomic region has no effect.

RANGE(<varname>, int min, int max)
This statement defines a nondeterministic choice over a discrete range of numeric variable values. The parameter *<varname>* must denote a variable name that is valid in the current scope. The parameters *min* and *max* describe the upper and lower border of the value range. Internally, the presence of a RANGE-statement corresponds to expanding a state to have $max - min$ successors. A RANGE statement must not appear within an atomic region. Also, the statement currently only works for *int* variables.

VASSERT(bool e)
This statement defines a local property. When, during program execution, VASSERT(e) is encountered, *StEAM* checks if the corresponding system state satisfies expression *e*. If *e* is violated, the model checker provides the user with the trail leading to the error and stops the search.

VLOCK(void * r)
A thread can request exclusive access to a resource by a VLOCK. This statement takes as its parameter a pointer to an arbitrary base type or structure. If the resource is already locked, the thread must interrupt its execution until the lock is released. If a locked resource is requested within an atomic region, the state of the executing thread is reset to the beginning of the region.

[9] A following semicolon is optional.

VUNLOCK(void * r)
Unlocks a resource making it accessible for other threads. The executing thread must be the holder of the lock. Otherwise this is reported as an access violation, and the error trail is returned.

5.4 Applicability

StEAM offers new possibilities for model checking software in several phases of the design process. In the design phase we can check, if our specification satisfies the required properties. Rather than using a model written in the input language of a model checker like e.g. SPIN, the developers can provide a test implementation written in same programming language as the end product, namely C/C++. On the one hand this eliminates the danger of missing errors that do not occur in the model but in the actual program. On the other hand, it helps to save the time used to search for so-called *false positives*, i.e. errors that occur due to an inconsistent model of the actual program.
More importantly, our tool is applicable in the testing phase of an actual implementation. Model checking has two major advantages over pure testing. First, it will not miss subtle errors, since the entire state space of the program is explored. Second, by providing an error trail, model checking gives an important hint for finding the source of the error instead of just claiming, that an error exists.

6 Experimental Results

We have made experiments with three different models, to measure the performance of our tool. The first model is the implementation of the dining philosophers, we presented in Section 3.3 which is analogous to the JAVA-version used in [8]. The second model implements an algorithm for the leader election protocol. Here an error was seeded, that assigns the same ID to each node and in turn can cause more than one process to be elected as the leader. Both of the above models are scalable to an arbitrary number of processes. The third model is a C++ conversion of the bounded buffer as introduced in [12]. The model is scalable in the number of tokens transmitted from the producer to the consumer. The search algorithms considered are breadth first search (BFS) and depth first search (DFS). The experiments have been done on a Linux based PC with AMD Athlon XP 2200+ processor, 1 GB RAM and 1802 MHz clock speed. Table 1 shows the results.

Model	Scale	Search Alg.	Memory	States	Trail-length	Time
philo	2	DFS	98642K	50	45	0.23
philo	2	BFS	99811K	265	14	0.25
philo	3	DFS	100752K	243	213	0.26
philo	3	BFS	133M	4337	20	1.01
philo	4	DFS	112M	1225	1065	0.43
philo	4	BFS	910M	70043	26	38.24
philo	5	DFS	199M	6342	5499	5.00
philo	5	BFS	mem. exceeded	-	-	
philo	6	DFS	745M	33038	28629	116.80
philo	7	DFS	mem. exceeded	-	-	
leader	2	DFS	98582K	60	50	0.23
leader	2	BFS	103372K	1740	35	0.37
leader	3	DFS	99011K	92	73	0.23
leader	3	BFS	101936K	36298	46	6.09
leader	4	DFS	99494K	92	73	0.23
leader	4	BFS	mem. exceeded	-	-	
leader	20	DFS	107443K	92	73	0.23
leader	40	DFS	115M	92	73	0.89
leader	80	DFS	137M	92	73	3.04
leader	500	DFS	518M	92	73	116.81
leader	700	DFS	mem. exceeded	-	-	
b. buffer	10	DFS	103525K	997	985	0.30
b. buffer	10	BFS	424M	47843	237	14.16
b. buffer	20	DFS	106M	1927	1905	0.46
b. buffer	20	BFS	1037M	111023	497	191.00
b. buffer	21	DFS	106M	2020	1997	0.46
b. buffer	21	BFS	1109M	117341	523	157.59
b. buffer	22	DFS	107M	2113	2089	0.51
b. buffer	22	BFS	mem. exceeded	-		-
b. buffer	40	DFS	117M	3787	3745	0.94
b. buffer	80	DFS	146M	7507	7425	2.67
b. buffer	160	DFS	224M	14947	14785	10.74
b. buffer	320	DFS	382M	29827	29505	62.16
b. buffer	400	DFS	625M	37267	36865	136.39
b. buffer	480	DFS	mem. exceeded	-		-

Table 1. Experimental results.

The first four columns indicate the tested model (Model), the scale option (Scale), the search algorithm (Search Alg.) and the amount of required memory (Memory). Note, that the latter includes some memory overhead like 64M which is reserved for the hash table. The last three columns show the number of visited states (States), the length of the error trail, and the required time in seconds (Time). The search is performed until an error state was found or until the model checker runs out of memory. In all three models, there is a significant difference between DFS and BFS. As one would expect, DFS finds error states for higher instances than BFS while having lower memory requirements. As an obvious drawback the error trails returned by DFS are longer than that of BFS. This is most significant for the dining philosophers. Even with merely four processes, DFS returns a trail with length 1065, which will hardly be traceable by any user. In contrast, BFS returns a trail, which is only 26 steps long. Also, Table 1 reveals two interesting phenomena. The first being, that in the leader election protocol and bounded buffer DFS can find errors even with the scale option set to a very high number. In contrast, DFS fails to find a deadlock in the dining philosophers with merely seven processes. The second phenomenon is, that for three or more processes, the trail length and number of visited states does not increase in the leader election problem. The reason for both of these phenomena is presumably twofold. First, we look at the way, how the models are scaled. For the bounded buffer, an increase of the scale option does not add new components to the state description, it merely widens the range of variable values. As a consequence, the size of the state space grows linearly, instead of exponentially. However, this does not seem to apply for the leader election protocol. Since this model is scaled by the number of processes, we also have an exponential increase in the size of the state space. This is confirmed by the fact, that BFS fails for instance sizes with merely four processes. Here, the second factor mentioned above comes into play, namely the error seed we use. For the simple seed, which assigns the same ID to each node, the number of actions required to trigger the error does not increase with the number of processes. In fact it suffices, that two processes send and receive one message. This happens quickly during depth first exploration, thus the number of explored states is also small. When having a closer look at the model checking process, we can observe, that DFS immediately finds an error after all processes have been initialized and failure implies that the initialization of the processes exceeds the amount of available memory. Another issue that may concern us is, that for the bounded buffer even the optimal error trail returned by BFS is relatively long for a small number of tokens. For example, in the case of 20 tokens we get an optimal trail with length 497. However, when inspecting the trail, we realize, that most of it is composed of a short sequence of executions, which is repeated over and over and corresponds to: “Sender sends a token, Consumer receives a token”. The implications of these observations for future work are as follows. First, a challenging model requires a good error seed, which assures, that either the difficulty to find the error or the length of the corresponding error trail grows with the scaling factor. Second, the length of the error trail for a scaling as used in the bounded buffer should be constant for an arbitrary scale option. This should be achieved either by a compact representation of the trail or an appropriate abstraction method.

7 Related Work

In this section, we discuss other projects, that deal with model checking or testing of software.

CMC[10] [17] is a model checker, which checks C and C++ implementations directly by generating the state space of the analyzed system during execution. *CMC* has been used to check correctness properties of network protocols, mainly the AODV[11] routing protocol[12] for ad-hoc networks. The checked correctness properties mentioned by the authors are assertion violations, a global invariant check avoiding routing loops, sanity checks on table entries and messages and memory errors (allocation errors, memory leaks). CMC is specialized to event based systems supporting process communication through shared memory. The direct successors of a given state are generated by calling all possible event handlers from the given state. CMC is capable to detect an error between two events and to state the error type (kind of violation). The only witness of an error is the sequence of events that have been processed before the error occurred. In contrast to *CMC*, *StEAM* keeps track of any executed code instruction. This gives more information than the sequence of events. A sequence of events does not lead straight forward to the executed source lines, while a *StEAM* error trail states every executed source line. This is the same basic information, a common debugger offers. To control the behavior of the dynamic memory allocation of `malloc`, `malloc` is overloaded, such that processes with the same sequence of calls to `malloc` have the same memory map for all allocated variables. *StEAM* identifies memory allocation and access by directly interpretating the machine code.

Verisoft [9] uses the same approach as *CMC*. A scheduler emulates the process environment and calls the different event handlers to generate all possible sequences of events. In contrast to *CMC*, *Verisoft* does not store all processed states in a hash table. They combine persistent sets and sleep sets, that refer to a notion of independency of transitions to restrict search. This approach is advantageously, if the search graph is finite and acyclic. To avoid infinite search for arbitrary search graphs, the search is depth-bounded. This approach minimizes memory usage, but may lead to repeated computation of same states that are not identified to be equal. The limitation of the search-depth can miss the detection of errors.

In [3], semantics are described to translate sC++ source code into Promela - the input language of the model checker Spin [15]. The language sC++ is an extension of C++ with concurrency. A lot of simplifying assumptions are made for the modeling process. For example, only basic types are considered, no structures, no type definitions, no pointers.

A similar approach is made by the tool *AX* (Automaton eXtractor) [16]. Here, Promela models can be extracted from C source code at a user defined level of abstraction.

[10] Abbreviation for C Model Checker
[11] Ad-hoc On-demand Distance Vector
[12] The result are 34 errors in the protocol.

BLAST (Berkley Abstraction Software Toolkit) is a software model checker for C programs, which is based on the property-driven construction and model checking of software abstractions [14]. The tools takes as its input a C program and a safety monitor written in C. The verification process is based upon counter example driven refinement: Starting from an abstract model of the program as a pushdown automaton, the tool checks, if the model fulfills the desired property. If an error state is found, BLAST automatically checks, if the abstract counterexample corresponds to a concrete counterexample in the actual program. If this is not the case, an additional set of predicates is chosen to build a more concrete model of the program and the property is checked anew.

Counterexample driven refinement is also used by the SLAM [2] tool. Here a boolean abstraction of the program is constructed. Then a reachability analysis is performed on the boolean program. Afterwards, additional predicates are discovered to define the boolean program - if necessary.

The Java PathFinder (JPF), is a model checker for concurrent Java programs. In its first versions (JPF1), the tool was merely a converter from Java source code to Promela [12,13]. The second generation[13] - JPF(2) - [18] uses a custom made Java Virtual Machine (JVM) to model check the compiled code of a program. Using this technique, JPF constitutes the first software model checker, which does not rely on an abstract model of the checked program. JPF also provides a set of heuristics, which can significantly accelerate the search for errors and ensure short, comprehensible counter examples [10]. An easy interface allows the implementation of user defined heuristics (see e.g. [8]).

8 Summary and Outlook

This paper gives a first presentation of StEAM, an assembly-level C++ model checker. We give an insight into the structure and working method of our tool. The purpose of the tool is to show, that the verification of actual C++ programs is possible without generating an abstract model of the source code. The experimental results show, that *StEAM* is in fact able to verify simple programs, or find errors in such programs. The main limitations currently lie in the high memory requirements. We will further improve the memory efficiency of the the state representation by an incremental storage of the state components. However, such methods can merely result in a linear decrease of the memory requirements, and will not be overly useful to cope with the state explosion problem. So the one of the main foci of current research lies in the integration of methods that make the latter less obstructive. These methods will include abstraction and the application of directed

[13] Since JPF1 is no longer under development, the term 'JPF' always refers to the second version of the model checker.

search methods with appropriate heuristics such as structural heuristics [10,6] and trail directed heuristics [5,8]. We will also integrate variations of DFS such as iterative deepening search. StEAM currently supports assertions for the definition of properties. This already allows testing for safety properties and invariants, which may be sufficient in many cases. However, for the verification of more complex properties, support for temporal logics like LTL is desirable. We will focus on how the functionality of such logics can be implemented in a straightforward manner, that is accessible for practitioners. Also, we will add automatic detection of illegal memory access. In the long term we will also consider, how the generic approach of *StEAM* can be generalized to other compiler-based programming languages such as Pascal.

Acknowledgements We thank Stefan Edelkamp for proofreading. Tilman Mehler is supported by the DFG project *Directed Model Checking with Exploration Algorithms of Artificial Intelligence*.

References

1. G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Doclady*, 3:1259–1263, 1962.
2. T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002.
3. T. Cattel. Modeling and Verification of sC++ Applications. In *4th Internatinal Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 232–248, 1998.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
5. S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
6. S. Edelkamp and P. Leven. Directed automated theorem proving. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 2514 of *LNAI*, pages 145–159. Springer, May 2002.
7. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.
8. S. Edelkamp and T. Mehler. Byte Code Distance Heuristics and Trail Direction for Model Checking Java Programs. In *2nd Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.
9. P. Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.
10. A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21. ACM Press, 2002.
11. J. Hatcliff and O. Tkachuck. The Bandera Tools for Model-checking Java Source Code: A User's Manual. Technical report, Kansas State University, 1999.

12. K. Havelund. *Java PathFinder User Guide.* NASA Ames Research Center, Moffet Field, CA, USA, 1999.
13. K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. Technical report, NASA Ames Research Center and Recom Technologies, 1999.
14. A. Henzinger, R. Jhala, R. Majumdar, and G. Stutre. Software Verification with BLAST. In *10th SPIN Workshop on Model Checking Software (SPIN), Lecture Notes in Computer Science 2648, Springer-Verlag*, pages 235–239, 2003.
15. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
16. G. J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885, pages 131–147. Springer-Verlag, 2000.
17. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*. USENIX Association, December 2002.
18. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. Post-CAV Workshop on Advances in Verification, (WAVe), 2000.