# Directed Error Detection in C++ with the Assembly-Level Model Checker StEAM

Peter Leven[1] and Tilman Mehler[2] and Stefan Edelkamp[2]

[1] Institut für Informatik, Georges-Köhler Allee, Geb. 51,
Albert-Ludwigs-Universität Freiburg, Germany
leven@informatik.uni-freiburg.de
[2] Fachbereich Informatik, Universität Dortmund, Baroper Str. 301,
44221 Dortmund, Germany
{tilman.mehler,stefan.edelkamp}@cs.uni-dortmund.de

**Abstract.** Most approaches for model checking software are based on the generation of abstract models from source code, which may greatly reduce the search space, but may also introduce errors that are not present in the actual program.
In this paper, we propose a new model checker for the verification of native c++-programs. To allow platform independent model checking of the object code for concurrent programs, we have extended an existing virtual machine for c++ to include multi-threading and different exploration algorithms on a dynamic state description.
The error reporting capabilities and the lengths of counter-examples are improved by using heuristic estimator functions and state space compaction techniques that additionally reduce the exploration efforts.
The evaluation of four scalable simple example problems shows that our system *StEAM*[3] can successfully enhance the detection of deadlocks and assertion violations.

## 1 Introduction

Model checking [4] refers to exhaustive exploration 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 [3,13,18,23] exploit model checking for the verification of actual programs written in e.g. Java or c.

Most of these approaches rely 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 [17]). The main advantage of abstract models is the reduction of state space.

Some model checkers - e.g. dSpin [6] - also consider dynamic aspects of computer programs, like memory allocation and dynamic object creation. These aspects must be mapped to the respective description language. If

[3] State Exploring Assembly Model Checker

this is not carefully addressed, the model may not correctly reflect all aspects of the actual program. Moreover, some of these approaches require manual intervention of the user which means that one has to be familiar with the input language of the model checker.

A different approach is considered in the Java model checker JPF. Instead of extracting a model from the source code, JPF [24] uses a custom-made `Java` virtual machine to check a program on the byte-code level. This eliminates the problem of an inadequate model of the program - provided that the virtual machine works correctly. The developers of JPF choose Java as the target programming language for several reasons [25]: First, Java features object-orientation and multi-threading in one language. Second, Java is simple. And third, Java is compiled to byte-code and hence the analysis can be done on byte-code level (as opposed to a platform-specific machine code). Also it was decided to keep JPF as modular and understandable to others as possible, sacrificing speed.

*StEAM*, the model checker presented in this paper, addresses a more general approach to such low-level program model checking. Based on a virtual processor, called the *Internet Virtual Machine* (IVM), the tool performs a search on machine-code compiled from a `c++` source. On one hand, this provides the option to model-check programs written in the industrial standard programming language, while, on the other hand, the generic approach is extendible to any compiler-based programming language with reasonable effort. Our method of reduction keeps state spaces small to compete with memory efficiency of other model checkers that apply model abstraction.
The architecture of *StEAM* is inspired by JPF. However the developers faced some additional challenges. First, there is no support for multi threading in standard `c++`. The language as well as the virtual machine had to be extended. Second, since the virtual machine is written in plain `c`, so is *StEAM*. Although this increases development time, we believe that the model checker will in the long term benefit from the increased speed of `c` compared to Java. Memory-efficiency is one of the most important issues of program model checking. There are various options for a time-space trade-off to save memory and explore larger state spaces. However, such techniques require that the underlying tool is fast. Moreover *StEAM* successfully ties the model checking algorithm with an existing virtual machine. A task thought impossible by the developers of JPF [25].

The paper is structured as follows. First, it introduces the architecture of the system. Next, it shows which extensions were necessary to enable program model checking, namely the storage of system states, the introduction of non-determinism through threads, and different exploration algorithms to traverse the state space in order to validate the design or to report errors. We illustrate the approach with a small example. The complex system state representation in *StEAM* is studied in detail. We introduce an apparent option for state space reduction and explain, why heuristics estimates accelerate the detection of errors and the quality of counter-examples. Exper-

iments show that *StEAM* effectively applies model checking of concurrent c++-programs. Finally we relate *StEAM* to other work in model checking, and conclude.

## 2 Architecture of the Internet C Virtual Machine

The *Internet C Virtual Machine* (ICVM) by Bob Daley aims at creating a programming language that provides platform-independence without the need of rewriting applications into proprietary languages like c♯ or Java. The main purpose of the project was to be able to receive precompiled programs through the Internet and run them on an arbitrary platform without recompilation. Furthermore, the virtual machine was designed to run games, so simulation speed was crucial.

*The Virtual Machine* The virtual machine simulates a 32-bit CISC CPU with a set of approximately 64,000 instructions. The current version is already capable of running complex programs at descend speed, including the commercial game Doom[4]. This is a strong empirical evidence that the virtual machine works correctly. Thus, dynamic aspects are carefully addressed. IVM is publicly available as open source[5].

*The Compiler* The compiler takes conventional c/c++ code and translates it into the machine code of the virtual machine. ICVM uses a modified version of the GNU C-compiler gcc to compile its programs. The compiled code is stored in ELF (Executable and Linking format), the common object file format for Linux binaries. The three types of file representable are object files, shared libraries and executables, but we will consider mostly executables.

*ELF Binaries* An ELF-binary is partitioned in sections describing different aspects of the object's properties. The number of sections varies depending on the respective file. Important are the DATA and BSS sections. Together, the two sections represent the set of global variables of the program.

The BSS section describes the set of non-initialized variables while the DATA section represents 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 non-initialized variables do not occupy space in the ELF file.

Space for initialized variables, however, is reserved in the DATA section of the object file, so accesses to variables directly affect the memory image of the ELF binary. Other sections represent executable code, symbol table etc., not to be considered for memorizing the state description.

[4] www.doomworld.com/classicdoom/ports
[5] ivm.sourceforge.net

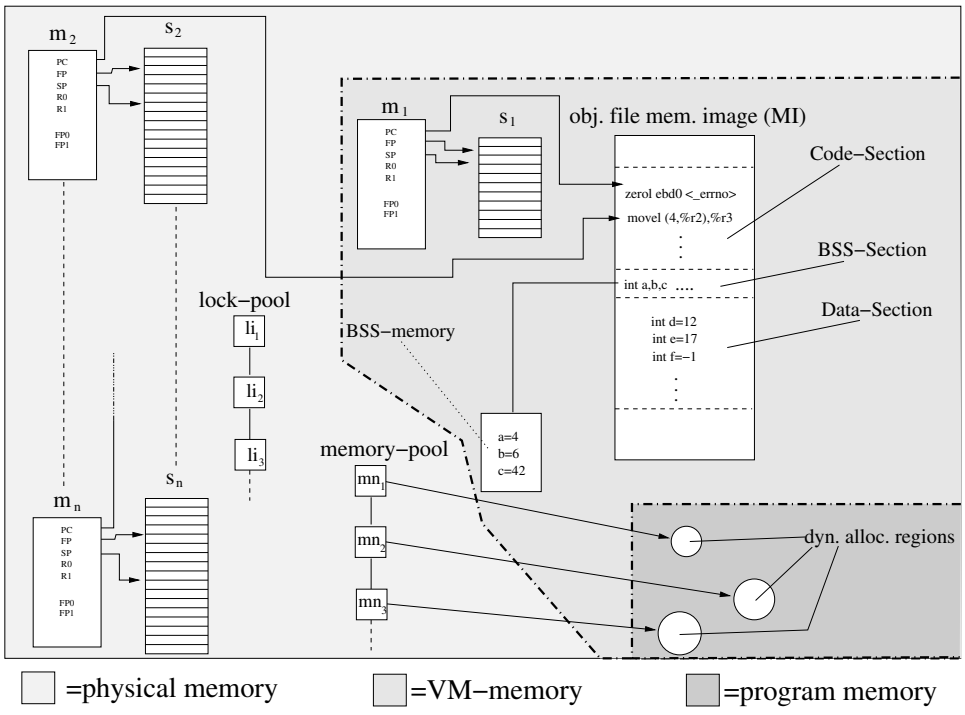


**Fig. 1.** System state of a *StEAM* program.

## 3 Multi-Threading

In the course of our project, the virtual machine was extended with multi-threading capabilities, 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 1 shows the components that form the state of a concurrent program for *StEAM*.

*System Memory Hierarchy* Memory is organized in three layers: Out-most is the physical memory which is only visible to the model checker. The subset *VM-memory* is also visible to the virtual machine and contains information about the main thread, i.e., the thread containing the main method of the program to check. The program memory forms a subset of the VM-memory and contains regions that are dynamically allocated by the program.

*Stacks and Machines* For $n$ threads, we have *stacks* $s_1, \ldots, s_n$ and *machines* $m_1, \ldots, m_n$, where $s_1$ and $m_1$ correspond to the *main* thread that is created when the verification process starts. Therefore, they reside in VM-memory. The machines contain the hardware registers of the virtual machine, such

as the program counter (PC) and the *stack* and *frame pointers* (SP, FP). Before the next step of a thread can be executed, the content of machine registers and stack must refer to the state immediately after the last execution of the same thread, or, if it is new, directly after initialization.

*Dynamic Process Creation* From the running threads, new threads can be created dynamically. Such a creation is recognized by *StEAM* through a specific pattern of machine instructions. *Program counters* (PCs) indicate the byte offset of the next machine instruction to be executed by the respective thread, i.e., they point to some position within the code-section of the object file's *memory image* (MI). MI also contains the information about the DATA and BSS sections. Note that (in contrast to the DATA section) the space for storing the contents of the variables declared in the BSS section lies outside the MI and is allocated separately.

*Memory- and Lock-Pool* The *memory-pool* is used by *StEAM* to manage dynamically allocated memory. It consists of an AVL-tree of entries (memory nodes), one for each memory region. They contain a pointer to address space which is also the search key, as well as some additional information such as the identity of the thread, from which it was allocated.

The *lock-pool* stores information about locked resources. Again an AVL-tree stores lock information.

## 4 Exploration

There is a core difference between the *execution* of a multi-threaded program and the *exploration* of its state space. In the first case, it suffices to restore machine registers and stack content of the executed thread.

To explore a program state space, the model checker must restore the state of DATA and BSS, as well as the memory and lock pool. Although *StEAM* does support program simulation, we consider only exploration.

*Special Command Patterns* On the programming level, multi-threading is realized through a base class `ICVMThread`, from which all thread classes must 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 the *start*-method and must contain the actual thread code. New commands e.g. `VLOCK` and `VUNLOCK` for locking have been integrated using macros. The compiler translates them to usual `c++`-code which does not influence the user-defined program variables. During program verification code patterns are detected in the virtual machine where special commands, like locking, are executed. This way of integration avoids manipulation of the compiler.

```
01. #include "IVMThread.h"
02. #include "MyThread.h"
04. extern int glob;
05.
06. class IVMThread;
07. MyThread::MyThread()
08.  :IVMThread::IVMThread(){
09. }
10. void MyThread::start() {
11.  run();
12.   die();
13. }
14.
15. void MyThread::run() {
16.   glob=(glob+1)*ID;
17. }
18.
19. void MyThread::die() {
20. }
21.
22. int MyThread::id_counter;
```

```
01. #include <assert.h>
02. #include "MyThread.h"
03. #define N 2
04.
05. class MyThread;
06. MyThread * t[N];
07. int i,glob=0;
08.
09. void initThreads () {
10.  BEGINATOMIC
11.    for(i=0;i<N;i++) {
12.      t[i]=new MyThread();
13.      t[i]->start();
14.     }
15.  ENDATOMIC
16. }
17.
18. void main() {
19.  initThreads();
20.  VASSERT(glob!=8);
21. }
```

**Table 1.** The source of the program `glob`.

*Example* Figure 1 shows a simple program `glob` which generates two threads from a derived thread class `MyThread`, that access a shared variable *glob*.

The `main` program calls 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`.

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 [17]. 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 example, we check the program against the expression `glob!=8`. Figure 2 shows the error trail of *StEAM*, when applied to `glob`. `Thread 1` denotes the main thread, `Thread 2` and `Thread 3` are two instances of `MyThread`. The returned error trail is easy to trace. First, 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 we have `glob=(3+1)*2=8`. After this, the line containing the `VASSERT`-statement is reached.

```
Step 1: Thread 1 - Line 10 src-file: glob.c -  initThreads
Step 2: Thread 1 - Line 16 src-file: glob.c -  main
Step 3: Thread 3 - Line 15 src-file: MyThread.cc - MyThread::run
Step 4: Thread 3 - Line 16 src-file: MyThread.cc - MyThread::run
Step 5: Thread 2 - Line 15 src-file: MyThread.cc - MyThread::run
Step 6: Thread 2 - Line 16 src-file: MyThread.cc - MyThread::run
Step 7: Thread 1 - Line 20 src-file: glob.c -  main
Step 8: Thread 1 - Line <unknown> src-file: glob.c -  main
```

**Fig. 2.** The error-trail for the 'glob'-program.

The assertion is only violated, if the `main` method of `Thread 3` is executed before the one of `Thread 2`. Otherwise, `glob` would take the values $0$, $2$, and $9$. By default, *StEAM* uses depth first search (DFS) for a program exploration. In general, DFS finds an error quickly while having low memory requirements. As a drawback, an 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, breadth-first search (BFS) and the heuristic search methods best-first (BF) and A$^*$ (see e.g. [9]).

*Detecting Deadlocks* *StEAM* automatically checks for deadlocks during a 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. A deadlock describes a state where all running threads wait for a lock to be released. A detailed example is given in [21].

*Hashing* *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. To improve memory efficiency, we fully store only those components of a state which differ from that of the predecessor state. If a transition leaves a certain component unchanged - which is often the case for e.g. the lock pool - only the reference to that component is copied to the new state. This has proven to significantly reduce the memory requirements of a model checking run. The method is similar to the *Collapse Mode* used in Spin [16]. However, instead of component indices, *StEAM* directly stores the pointers to the structures describing respective state components. Also, only components of the immediate predecessor state are compared to those of the successor state. A redundant storing of two identical components is therefore possible. Additional savings may be gained through reduction techniques like heap symmetry [19], which are subject to further development of *StEAM*.

## 5 Accelerating Error Detection

Our approaches to accelerate error detection are twofold. First, we reduce the state space of assembly-level program state exploration. Second, we invent heuristics, which accelerate error detection, especially the search for deadlocks.

Although *StEAM* can model check real c++ programs, it is limited in the size of problems it can handle. Unmodified c++ programs have more instructions than abstract models other model checker take as their input. The state space usually grows exponential in the number of threads including the number of executed machine instruction in each thread as a factor. For each instruction, all permutations of thread orders can occur and have to be explored.

*Lock and Global Compaction* The occurrence of an error does not depend on every execution order. An exploration of a single thread has to be interrupted only after lock/unlock or access to shared variables. Each access to local memory cells cannot influence the behaviour of other threads. Therefore, we realized two kinds of state reduction techniques.

The first one, called *lock* and *global compaction*, *lgc* for short, executes each thread until the next access to shared memory cells. Technically, this is performed by looking at the memory regions, that each assembly level instruction accesses and at lock instructions.

*Source Line Compaction* The second kind of exploration, *nolgc* for short, requires each source line to be atomic. No thread switch is allowed during execution of a single source line. This is not immediate, since each line of code correspond to a sequence of object code instruction.

The implication for the programmer is that infinite loops that e.g wait for change of a shared variable, are not allowed in a single source line. We expect the body of the loop to be unfolded in forthcoming source. The source line compaction is only sound with respect to deadlock detection, if read and write access as well as lock and unlock access to the same variable are not included in one line.

Both techniques reduce thread interleaving and link to the automated process of partial order reduction, which has been implemented in many explicit state model checking systems [20,7]. The difference is that we decide, whether or not a thread interleaving has to be considered by looking at current assembler instruction and the lock pool.

## 6 Directed Program Model Checking

Heuristics have been successfully used to improve error detection in concurrent programs, see e.g. [12,8]. States are evaluated by an estimator function, measuring the distance to an error state, so that states closer to the faulty

behavior have a higher priority and are considered earlier in the exploration process. If the system contains no error, there is no gain, the whole search space is enumerated. Compared to blind search, the only loss is due to additional computational resources for the heuristics.

*Most-Block and Interleaving* An appropriate example for the detection of deadlocks is the *most-block* heuristic. It favors states, where more threads are blocked. Another established estimate used for error detection in concurrent programs is the *interleaving* heuristic. It relies on maximizing the interleaving of thread executions [12].

In the following we consider new aspects to improve the design of estimator functions. In *StEAM* heuristics either realize single ideas or mixed ones, so we first start with three basic primitives: lock, shared variables and thread-id, followed by a treatment on how to combine them to higher-order functions.

*Lock* In the *lock* heuristic, we prefer states with more variables locks and more threads alive. Locks are obvious preconditions for threads to become blocked. Only threads that are still alive can get blocked in the future.

*Shared Variable* Finally, we considered the access to shared variables. We prefer a change of the active thread after a global read or write access. The objective is that after accessing a global variable, other threads are likely to be affected.

*Thread-Id* Threads are of equal class in many cases. If threads have equal program code and differ only in their *thread-id*, their internal behaviour is only slightly different. If the threads are ordered linear ascending according to their id, we may prefer the last one.

The *thread-id* heuristics can be seen as kind of a *symmetry reduction* rule, because we impose a preference ordering on similar threads, which sets a penalty to the generation of equivalent state generation. Symmetric reduction based on ordered thread-IDs and their PC values is e.g. analyzed in [2] and integrated into *dSpin*. One advantage compared to other approaches in symmetry reduction is, that we encoded the similarity measure into the estimator function. Therefore, the approach is more flexible and can be combined easily with other heuristics. Moreover, no explicit computation of canonical states takes place and the approach is not specialized to a certain problem domain.

*Favoring Patterns* Each thread has internal values and properties that we can use to define which thread execution we favor in an exploration step, e.g. thread-id (ID), PC, number of locked variables, number of executed instructions so far, and flags like *blocked* and *alive*. If *blocked* is set, the thread is waiting for a variable to be unlocked. If *alive* is unset, the thread will not be executed anymore: it is dead.

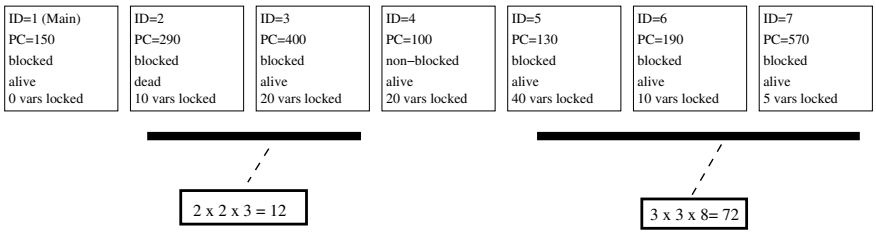


**Fig. 3.** Favoring patterns.

We select a subset of all possible components to define heuristics. The pattern for some relevant components for an example system state is shown in Figure 3. We call those patterns *favoring*, as they favor states in the exploration. In the following we explain one of the heuristics in use, namely the *pbb* heuristic.

If we add simply the number of all blocked threads in an heuristic, we obtain the mentioned most blocked heuristic. But many states have the same number of blocked threads. The number of equal states, that can be obtained from a given state by permuting thread-ids and reordering the threads can be exponential in the number of threads.

To favor only a few of equivalent states, one concept of favoring patterns we use are *neighbor groups*, maximal groups of consecutive threads having a certain property, in our example flag *blocked*. Additionally, we prefer systems states, where the *neighbor groups* are rightmost. We abbreviate *neighbor group* by *group*.

For a combined estimate for the entire state, we first square the size of each group. To express the preference for rightmost group, we multiply the obtained value with the largest thread-id in the group. Then the values for each *group* are added. In the example of Figure 3, we have value $12$ for the first group and value $72$ for the second group yielding the sum $94$.

To prefer states with more threads blocked, we additionally add the cubed number of the total of all blocked threads. For the maximum possible number of blocked threads this value is compatible with the previous sum. In our example five threads are blocked, so that the final preference value is $94 + 625 = 721$.

Table 2 summarizes the applied heuristics together with a brief description. Newly contributed heuristic start with *pba*, known ones refer to [12].

## 7 Experimental Results

An evaluation of *StEAM* has been performed on a Linux based PC (AMD Athlon XP 2200+ processor, 1 GB RAM, 1,800 MHz clock speed). The memory has been limited to 900 MB and the search time to 20 minutes.

| Heuristic | Abb. | Description |
|---|---|---|
| most blocked | mb | Counts the number of blocked threads |
| interleaving | int | The history of executed thread numbers is considered to prefer least recently executed threads |
| preferred alive & blocked | pba | Adds the cubed number of all blocked threads, the squared number of all alive threads and for each *group* of alive threads the squared number of threads weighted with the rightmost thread-id. |
| preferred blocked | pbb | Add the cubed number of all blocked threads, the squared number of all blocked threads and for each *group* of blocked threads the squared number of threads weighted with the rightmost thread-id. |
| read & write | rw | Prefers write access to shared variables and punishes read accesses without intermediate write access on each thread. |
| lock n' block | lnb | Count the number of locks in all threads and the number of blocked threads. |
| preferred locked 1 | pl1 | Counts the squared number of locks in all threads and the squared number of blocked threads. |
| preferred locked 2 | pl2 | Counts the squared number of locks in all threads and the squared number of *group*s of blocked and alive threads, where rightmost *group*s are preferred by weighting each *group* with the largest thread-id. |
| alternating read & write | aa | Prefers alternating read and write access. |

**Table 2.** Known (*mb,int*) and newly introduced heuristics.

*Models* We conducted experiments with four scalable programs, e.g. simple communication protocols. Even though the selection is small and contains not very elaborated case studies, the state space complexity of the programs can compare with the code fragments that are often considered for program model checking.

The first model is the implementation of a deadlock solution to the dining philosophers problem (*philo*) as described in [10,21].

The second model implements an algorithm for the leader election protocol (*leader*). Here an error was seeded, which 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++ implementation of the optical telegraph protocol (*opttel*), which is described in [15]. The model is scalable in the number of telegraph stations and contains a deadlock.

The fourth model is an implementation of a bank automata scenario (*cashit*). Several bank automata perform transaction on a global database (withdraw, request, transfer). The model is scalable in the number of automata. A wrong lock causes an access violation.

*Undirected Search* The undirected search algorithms considered are BFS and DFS. Table 3 shows the results. For the sake of brevity we only show the result for the maximum scale ($s$) that could be applied to a model with a specific combination of a search algorithm with or without state space compaction ($c$), for which an error could be found. We measure trail length ($l$), the search time ($t$ in 0.1s) and the required memory ($m$ in KByte). In the following $n$ denotes the scale factor of a model and $msc$ the maximal scale.

In all models BFS already fails for small instances. For example in the dining philosophers, BFS can only find a deadlock up to $n = 6$, while heuristic search can go up to $n = 190$ with only insignificant longer trails. DFS only has an advantage in *leader*, since higher instances can be handled, but the produced trails are longer than those of heuristic search.

*Heuristic Search* Table 4 and Table 5 depict the obtained results for directed program model checking.

The *int* heuristic, used with BF, shows some advantages compared to the undirected search methods, but is clearly outperformed by the heuristics which are specifically tailored to deadlocks and assertion violations. The *rw* heuristic performs very strong in both, *cashit* and *leader*, since the error in both protocols is based on process communication. In fact *rw* produces shorter error trails than any other method (except BFS).

In contrast to our expectation, *aa* performs poorly at the model *leader* and is even outperformed by BFS with respect to scale and trail length. For *cashit*, however, *aa*, leading to the shortest trail and *rw* are the only heuristic that find an error for $n = 2$, but only if the *lgc* reduction is turned off. Both of these phenomena of *aa* are subject to further investigation.

The lock heuristics are especially good in *opttel* and *philo*. With BF and *lgc* they can be used to a $msc$ of 190 (*philo*) and 60 (*opttel*). They outperform other heuristics with *nolgc* and the combination of A* and *lgc*. In case of A* and *nolgc* the results are also good for *philo* (n=6 and n=5; msc is 7).

According to the experimental results, the sum of locked variables in continuous block of threads with locked variable is a good heuristic measure to find deadlocks. Only the *lnb* heuristic can compare with *pl1* and *pl2* leading to a similar trail length. In case of *cashit pl2* and *rw* outperform most heuristics with BF and A*: with *lgc* they obtain an error trail of equal length, but *rw* needs less time. In the case of A* and *nolgc pl2* is the only heuristic which leads to a result for *cashit*. In most cases both *pl* heuristics are among the fastest.

The heuristic *pba* is in general moderate but sometimes, e.g. with A* and *nolgc* and *opttel* ($msc$ of 2) and *philo* ($msc$ of 7) outperforming. The heuristic *pbb* is often among the best, e.g. *leader* with BF and *lgc* ($n = 6$, $msc = 80$), *philo* with BF and *lgc* (n=150; $msc$ is 190) and *philo* with A* and *nolgc* ($n = 6$, $msc = 7$). Overall the heuristics *pba* and *pbb* are better suited to A*.

*Experimental Summary* Figure 4 summarizes our results. We measure the performance by extracting the fourth root of the product of trail length, pro-

| | | cashit | | | | leader | | | | opttel | | | | philo | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $c$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ |
| DFS | y | 1 | 97 | 151 | 134 | 80 | 1,197 | 247 | 739 | 9 | 8,264 | 214 | 618 | 90 | 1,706 | 166 | 835 |
| DFS | n | 1 | 1,824 | 17 | 96 | 12 | 18,106 | 109 | 390 | 6 | 10,455 | 221 | 470 | 20 | 14,799 | 463 | 824 |
| BFS | y | - | - | - | | 5 | 56 | 365 | 787 | 2 | 21 | 24 | 97 | 6 | 13 | 160 | 453 |
| BFS | n | - | - | - | | 3 | 66 | 30 | 146 | - | - | - | | 4 | 26 | 57 | 164 |

**Table 3.** Results with Undirected Search.

| | | cashit | | | | leader | | | | opttel | | | | philo | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SA | C | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ |
| pl1 | y | 1 | 97 | 150 | 134 | 5 | 73 | 89 | 285 | 60 | 602 | 210 | 770 | 190 | 386 | 166 | 622 |
| pl2 | y | 1 | 97 | 59 | 120 | 5 | 73 | 94 | 312 | 60 | 601 | 210 | 680 | 190 | 385 | 487 | 859 |
| mb | y | 1 | 97 | 150 | 134 | 80 | 1,197 | 247 | 737 | 9 | 6,596 | 160 | 518 | 150 | 750 | 213 | 862 |
| *lnb* | y | 1 | 97 | 150 | 134 | 5 | 73 | 86 | 263 | 60 | 602 | 211 | 763 | 190 | 386 | 168 | 639 |
| int | y | 1 | 98 | 91 | 128 | 5 | 57 | 395 | 789 | 2 | 22 | 12 | 98 | 8 | 18 | 193 | 807 |
| aa | y | 1 | 84 | 03 | 97 | 5 | 72 | 278 | 637 | - | - | - | | 6 | 14 | 208 | 449 |
| rw | y | 1 | 61 | 02 | 97 | 60 | 661 | 195 | 671 | 2 | 309 | 07 | 98 | 90 | 1,706 | 178 | 854 |
| pba | y | 1 | 97 | 90 | 121 | 80 | 1,197 | 244 | 740 | 9 | 6,596 | 161 | 513 | 90 | 450 | 43 | 234 |
| *pbb* | y | 1 | 97 | 150 | 134 | 80 | 1,197 | 244 | 741 | 9 | 6,596 | 160 | 510 | 150 | 750 | 210 | 860 |
| pl1 | n | 1 | 1,824 | 17 | 97 | 4 | 245 | 94 | 284 | 40 | 1,203 | 200 | 705 | 150 | 909 | 340 | 859 |
| pl2 | n | 1 | 792 | 09 | 97 | 4 | 245 | 168 | 376 | 40 | 1,202 | 193 | 751 | 150 | 908 | 327 | 857 |
| mb | n | 1 | 1,824 | 17 | 97 | 12 | 18,106 | 107 | 428 | 6 | 10,775 | 235 | 489 | 60 | 1,425 | 102 | 548 |
| *lnb* | n | 1 | 1824 | 17 | 97 | 4 | 245 | 92 | 284 | 40 | 1,203 | 197 | 719 | 150 | 909 | 344 | 856 |
| int | n | 1 | 1,824 | 17 | 97 | 3 | 68 | 37 | 139 | - | - | - | | 5 | 33 | 234 | 663 |
| aa | n | 2 | 328 | 86 | 146 | 3 | 132 | 28 | 139 | - | - | - | | 4 | 27 | 73 | 197 |
| rw | n | 2 | 663 | 147 | 463 | 40 | 1,447 | 104 | 470 | 2 | 494 | 68 | 206 | 20 | 14,799 | 469 | 822 |
| pba | n | 1 | 792 | 14 | 97 | 12 | 18,106 | 109 | 411 | 6 | 10,775 | 235 | 470 | 40 | 945 | 33 | 239 |
| *pbb* | n | 1 | 1,824 | 17 | 97 | 12 | 18,106 | 109 | 407 | 6 | 10,775 | 236 | 471 | 60 | 1,425 | 102 | 571 |

**Table 4.** Results with Best-First Search.

cessed states, time and memory (geometric mean of the arguments). We use a log-scale on both axes.

In the case of the leader election protocol, the advantage of compaction is apparent. The maximum possible scale almost doubles, if the compaction is used. The graphic shows that *rw* and DFS perform similar, finally DFS is a little bit better with lgc, but *rw* is much better with *nolgc*. BFS, *int*, and *aa* behave similar in both cases.

In *opttel*, the heuristics *lnb*, *pl1* and *pl2* are performing best. The *pl2* heuristic only starts to perform well with $n = 15$, before the curve has a high peak. It seems, that the preference of continuous blocks of alive or blocked thread has only a value, after increasing a certain scale, here 10. The *pab* and *pbb* heuristic perform similar up to an $msc$ of 9.

| | | cashit | | | | leader | | | | opttel | | | | philo | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $c$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ | $s$ | $l$ | $t$ | $m$ |
| pl1 | y | - | - | - | | 5 | 56 | 345 | 711 | 3 | 32 | 39 | 215 | 190 | 386 | 166 | 631 |
| pl2 | y | 1 | 61 | 2,781 | 387 | 5 | 56 | 358 | 760 | 2 | 21 | 03 | 98 | 190 | 385 | 482 | 860 |
| mb | y | - | - | - | | 5 | 56 | 397 | 769 | 2 | 22 | 19 | 98 | 7 | 16 | 111 | 371 |
| *lnb* | y | - | - | - | | 5 | 56 | 342 | 719 | 2 | 22 | 15 | 98 | 8 | 18 | 82 | 286 |
| int | y | - | - | - | | 5 | 57 | 406 | 785 | 2 | 22 | 13 | 98 | 7 | 16 | 177 | 629 |
| aa | y | - | - | - | | 5 | 61 | 385 | 744 | 2 | 22 | 41 | 171 | 6 | 14 | 208 | 449 |
| rw | y | 1 | 61 | 1958 | 310 | 7 | 78 | 163 | 589 | 2 | 22 | 24 | 98 | 6 | 14 | 208 | 449 |
| pba | y | - | - | - | | 5 | 56 | 378 | 648 | 3 | 32 | 130 | 424 | 60 | 122 | 191 | 628 |
| *pbb* | y | - | - | - | | 5 | 56 | 396 | 777 | 3 | 32 | 195 | 573 | 60 | 122 | 193 | 634 |
| pl2 | n | 1 | 136 | 10,568 | 614 | 3 | 66 | 32 | 141 | - | - | - | | 5 | 38 | 172 | 485 |
| mb | n | - | - | - | | 3 | 66 | 32 | 151 | - | - | - | | 4 | 27 | 39 | 155 |
| rw | n | - | - | - | | 3 | 66 | 21 | 97 | - | - | - | | 4 | 27 | 71 | 199 |
| pba | n | - | - | - | | 3 | 66 | 29 | 138 | 2 | 63 | 547 | 845 | 7 | 51 | 424 | 872 |
| *pbb* | n | - | - | - | | 3 | 66 | 32 | 139 | - | - | - | | 6 | 45 | 187 | 431 |

**Table 5.** Results for A*.

In *philo*, the heuristics *pl1*, *pl2*, *pba*, *pbb* are performing best. If only BF is considered, the heuristic *lnb* behaves similar than *pl1* and *pl2*. Again, *pl2* has an initial peak. DFS is performing well to $msc$ of 90.

In the experiments the new heuristics show an improvement in many cases. In the case of deadlock search the new lock and block heuristics are superior to *most blocked*. The *lgc* compaction often more than doubles the maximum possible model scale.

## 8 Related Work

We discuss other projects, that deal with model checking or software testing.

*CMC* [22], the `c` Model Checker, checks `c` and `c++` implementations directly by generating the state space of the analyzed system during execution. *CMC* has mainly been used to check correctness properties of network protocols The checked correctness properties are assertion violations, a global invariant check avoiding routing loops, sanity checks on table entries and messages and memory errors. CMC is specialized to event based systems supporting process communication through shared memory. The successor states 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 kind of violation. The only witness of an error is the sequence of processed 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. To control the behaviour of the dynamic memory allocation, `malloc` is overloaded, such that processes with the same sequence of calls to `malloc` have

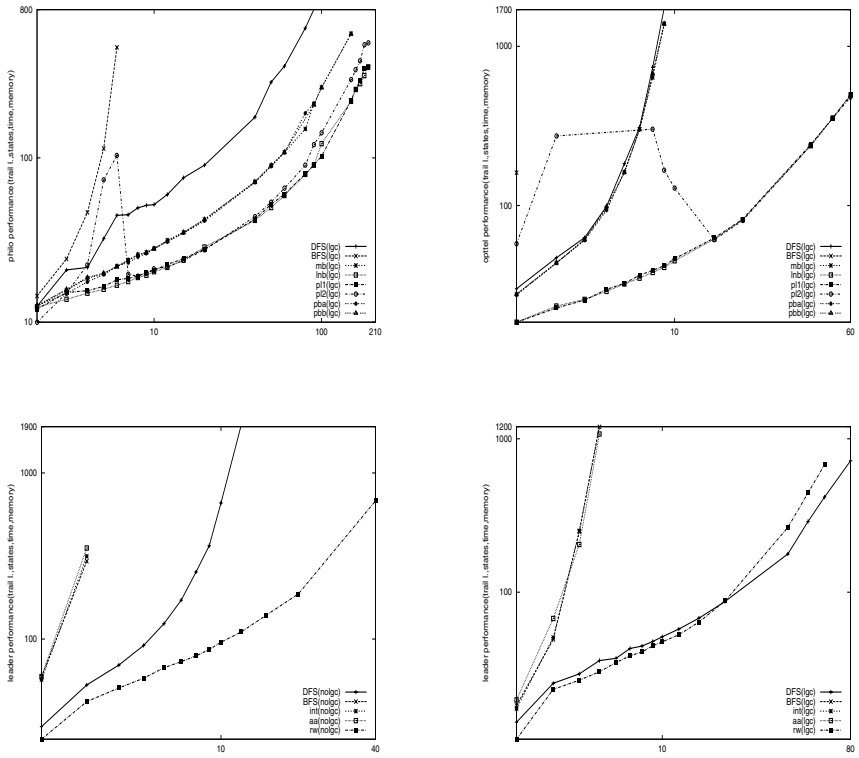


**Fig. 4.** Performance for philo/lgc, opttel/lgc, leader/nolgc, leader/lgc. x-axes denote scale, y-axes denote the geometric mean of trail length, states, time and memory.

the same memory map for all allocated variables. *StEAM* identifies memory allocation and access by directly interpreting the machine code.

*Verisoft* [11] uses the same approach as *CMC*. A scheduler emulates the process environment and calls event handlers to generate all possible sequences of events. In contrast to *CMC*, *Verisoft* does not store all processed states in a hash table. It combines persistent sets and sleep sets, that refer to a notion of independency of transitions to restrict search. This 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 identical

states that are not identified as such. The limitation of the search-depth can miss the detection of errors.

*sC++ and AX* In [3], semantics are described to translate sC++ source code into Promela - the input language of the model checker Spin [17]. The language sC++ is an extension of C++ with concurrency. Many simplifying assumptions are made for the modeling process: only basic types are considered, no structures, no type definitions, no pointers. A similar approach is made by the tool *AX* (Automaton eXtractor) [18]. Here, Promela models can be extracted from C source code at a user defined level of abstraction.

*BLAST* [14], the *Berkeley Abstraction Software Toolkit* is a model checker for C programs, which is based on the property-driven construction and model checking of software abstractions. 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 and the property is checked anew.

*SLAM* [1] also uses counter-example driven refinement. 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.

*Bandera* [5] constitutes a multi-functional tool for Java program verification. Bandera is capable of extracting models from Java source code and converting them to the input language of several well known model checkers such as Spin or SMV. As one very important option to state space reduction, Bandera allows to *slice* source code.

*Bogor* [23] is a model checking framework with an extendible input language for defining domain-specific state space encodings, reductions and search algorithms. It allows domain experts to build a model checker optimized for their specific domain without in-depth knowledge about the implementation of a specific model checker. The targeted domains include code, designs and abstractions of software layers. Bogor checks systems specified in a revised version of the BIR format, which is also used in Bandera [5,13].

## 9 Conclusion

This paper introduces *StEAM*, an assembly-level C++ model checker. We give insight into the structure and working 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.

Our approach of directed program model checking outperforms undirected search with respect to trail length and maximum scale. We further extended the set of heuristics in model checking that are specifically tailored to find deadlocks and assertion violations. Many of the newly invented heuristics perform better than known heuristics. One option are *favoring pattern*, heuristics that relate to *symmetry reduction*. Another contribution are *lgc*s, that relate to *partial order reduction*. Both approaches encode pruning options in form of state preference rules and significantly improves the performance with respect to most search methods.

*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. Subsequent work 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 future we will also consider new models, heuristics and search methods, e.g. we will integrate variations of DFS such as iterative deepening and related search methods to reduce the memory requirements.

***Acknowledgements*** Tilman Mehler is supported by DFG, in the project *Directed Model Checking with Exploration Algorithms of Artificial Intelligence*. Stefan Edelkamp is supported by DFG, in the project *Heuristic Search*. We thank Bob Daley for his friendly support concerning ICVM.

## References

1. T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002.
2. D. Bosnacki, D. Dams, and L. Holenderski. A Heuristic for Symmetry Reductions with Scalarsets. In *International Symposium on FME: Formal Methods for Increasing Software Productivity*, pages 518–533, 2001.
3. T. Cattel. Modeling and Verification of sC++ Applications. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 232–248, 1998.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
5. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubachand, and H. Zheng. Extracting Finite-state Models from Java Source Code. In *International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.
6. C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Model Checking Software (SPIN)*, pages 261–276, 1999.
7. M. Dwyer, J. Hatcliff, V. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. *To appear in Formal Methods in System Design Journal (FMSD)*, 2004.

8. 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.
9. S. Edelkamp and P. Leven. Directed Automated Theorem Proving. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 145–159, 2002.
10. 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.
11. P. Godefroid. Model Checking for Programming Languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.
12. A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 12–21, 2002.
13. J. Hatcliff and O. Tkachuck. The Bandera Tools for Model-checking Java Source Code: A User's Manual. Technical report, Kansas State University, 1999.
14. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Model Checking Software (SPIN)*, pages 235–239, 2003.
15. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
16. G. J. Holzmann. State Compression in SPIN. In *Third Spin Workshop*, Twente University, The Netherlands, 1997.
17. G. J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
18. G. J. Holzmann. Logic Verification of ANSI-C code with SPIN. In *Model Checking Software (SPIN)*, pages 131–147, 2000.
19. R. Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *International Conference on Automated Software Engineering (ICSE)*, pages 26–29, 2001.
20. A. Lluch-Lafuente, S. Leue, and S. Edelkamp. Partial Order Reduction in Directed Model Checking. In *Model Checking Software (SPIN)*, pages 112–127, 2002.
21. T. Mehler and P. Leven. *Introduction to StEAM - An Assembly-Level Software Model Checker*. Technical report, University of Freiburg, 2003.
22. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
23. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In *European Software Engineering Conference (ESEC)*, pages 267–276, 2003.
24. 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.
25. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *International Conference on Automated Software Engineering (ICSE)*, pages 3–12, 2000.