Challenges and Applications of Assembly-Level Software Model Checking

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
an der Universität Dortmund
am Fachbereich Informatik
von

Tilman Mehler

Dortmund

2005

Tag der mündlichen Prüfung:	
Dekan/Dekanin:	Prof. Bernhard Steffen
Gutachter:	Dr. Stefan Edelkamp, Prof. Katharina Morik.
Note:	☐ ausgezeichnet ☐ sehr gut ☐ gut ☐ genügend

Für meine Eltern.

Acknowledgements

Hereby, I would like to thank my advisor Dr. Stefan Edelkamp, for his excellent guidance. Without his various original ideas, his broad theoretical and practical knowledge, his reviews and moral support, this thesis would not exist. Moreover, I thank Prof. Dr. Katharina Morik and Dr. Doris Schmedding who agreed to review my thesis. Also, I want to thank Shahid Jabbar for his support and his reviews. Further acknowledgements must go to the German Research Society (DFG) who financially supported my research through the project *Directed Model Checking with Exploration Algorithms of Artificial Intelligence*. Last, but not least, I would like to thank my girlfriend Frida as well as my friends and family for their moral support.

Summary

This thesis addresses the application of a formal method called *Model Checking* to the domain of software verification. Here, exploration algorithms are used to search for errors in a program. In contrast to the majority of other approaches, we claim that the search should be applied to the actual source code of the program, rather than to some formal model.

There are several challenges that need to be overcome to build such a model checker. First, the tool must be capable to handle the full semantics of the underlying programming language. This implies a considerable amount of additional work unless the interpretation of the program is done by some existing infrastructure. The second challenge lies in the increased memory requirements needed to memorize entire program configurations. This additionally aggravates the problem of large state spaces that every model checker faces anyway. As a remedy to the first problem, the thesis proposes to use an existing virtual machine to interpret the program. This takes the burden off the developer, who can fully concentrate on the model checking algorithms. To address the problem of large program states, we call attention to the fact that most transitions in a program only change small fractions of the entire program state. Based on this observation, we devise an incremental storing of states which considerably lowers the memory requirements of program exploration. To further alleviate the per-state memory requirement, we apply *state reconstruction*, where states are no longer memorized explicitly but through their generating path. Another problem that results from the large state description of a program lies in the computational effort of hashing, which is exceptionally high for the used approach. Based on the same observation as used for the incremental storing of states, we devise an incremental hash function which only needs to process the changed parts of the program's state. Due to the dynamic nature of computer programs, this is not a trivial task and constitutes a considerable part of the overall thesis.

Moreover, the thesis addresses a more general problem of model checking - the *state explosion*, which says that the number of reachable states grows exponentially in the number of state components. To minimize the number of states to be memorized, the thesis concentrates on the use of heuristic search. It turns out that only a fraction of all reachable states needs to be visited to find a specific error in the program. Heuristics can greatly help to direct the search forwards the error state. As another effective way to reduce the number of memorized states, the thesis proposes a technique that skips intermediate states that do not affect shared resources of the program. By merging several consecutive state transitions to a single transition, the technique may considerably truncate the search tree.

The proposed approach is realized in *StEAM*, a model checker for concurrent C++ programs, which was developed in the course of the thesis. Building on an existing virtual machine, the tool provides a set of blind and directed search algorithms for the detection of errors in the actual C++ implementation of a program. StEAM implements all of the aforesaid techniques, whose effectiveness is experimentally evaluated at the end of the thesis.

Moreover, we exploit the relation between model checking and planning. The claim is, that the two fields of research have great similarities and that technical advances in one fields can easily carry over to the other. The claim is supported by a case study where StEAM is used as a planner for concurrent multi-agent systems.

The thesis also contains a user manual for StEAM and technical details that facilitate understanding the engineering process of the tool.

Contents

Chapter 1

Introduction

1.1 Motivation

The increasing role of software in safety-critical areas imposes new challenges to computer science. Software systems often contain subtle errors, which remain undetected by manual code inspection or pure testing. Errors, that when they eventually arise after deployment may cause heavy financial damage or even the loss of human lives. We name a few examples.

1.1.1 Deadly Software Errors

The Lufthansa Warsaw Accident

In 1993 the Lufhansa flight DLH 2904 from Frankfurt to Warsaw ended in a disaster when the aircraft tried to land at its destination airport. At that time, the software which controls the thrust reversers would not allow them to activate unless the aircraft's gears were down and the aircraft's weight was pressing on the wheels. As an unfortunate coincidence, heavy wind lifted up the aircraft resulting in a lack of compression on the left gear leg. As a result, the thrust reversers were started with a delay of 9 seconds, which caused the aircraft to leave the runway and crash into the surrounding field. One crew member and one of the passengers died.

Therac-25 Accident The Therac-25 was a medical linear accelerator that could be used to destroy tumors while inflicting relatively little damage to the surrounding tissue. A complex bug in the control software caused six cases of heavy radiation overdoses between 1985 and 1987, causing death and severe injuries to patients.

1.1.2 Expensive Software Errors

Ariane 5 [1]

"On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency (ESA) exploded just forty seconds after its lift-off from Kourou, French Guyana at an

[1]from Scientific Digital Visions, Inc - http://www.dataenabled.com/

altitude of 3,700 meters. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million ... It turned out that the cause of the failure was a software error in the inertial reference system. Specifically, a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,768, the largest integer storable in a 16 bit signed integer, and thus the conversion failed. ...”

Mars Climate Orbiter [2]

“The Mars Climate Orbiter is part of a series of missions in a long-term program of Mars exploration. ... the spacecraft was lost on September 23, 1999 while entering the orbit around Mars. After an internal peer review at JPL [Jet Propulsion Laboratory], several problems were discovered. The review indicated a failure to recognize and correct an error in a transfer of data between the Mars Climate Orbiter spacecraft team in Colorado and the mission navigation team in California. In particular, there was a failed translation of English units into metric units in a segment of ground-based, navigation-related mission software. As a result, the spacecraft was lost. In order to prevent future problems, all data and engineering processes related to the Mars Polar Lander were scrutinized.”

Note: $150M lost.

Another popular example is the software bug of the *Mars path finder* explained in section 2.4.

1.1.3 Software Engineering Process

Many industrial software projects obey the so-called waterfall model, which distinguishes six phases: requirements analysis, design, implementation, testing, integration and maintainance. These are illustrated in Figure 1.1.

The model suggests an iterative process of software development, which allows the project to switch between neighbouring phases. For instance, the software engineers may fall back from the testing phase to the implementation phase, if testing exposes an error and they may further fall back to the design phase, if during source investigation it turns out that the cause is a general flaw in the system’s design, rather than a simple implementation error. An obvious observation is, that correcting an error gets more expensive if it occurs at a later phase of the development process. In the design phase of the project, the required functionalities are usually specified with specialized modeling languages, such as Z. These languages provide a standardized notation for the behavior of program functions, such that their correctness can be verified through manual mathematical proofs. Then, the formal specification is implemented in the targeted programming language. Before the software is integrated, the correct functionality is tested in a simulated environment.

[2] from Scientific Digital Visions, Inc - http://www.dataenabled.com/

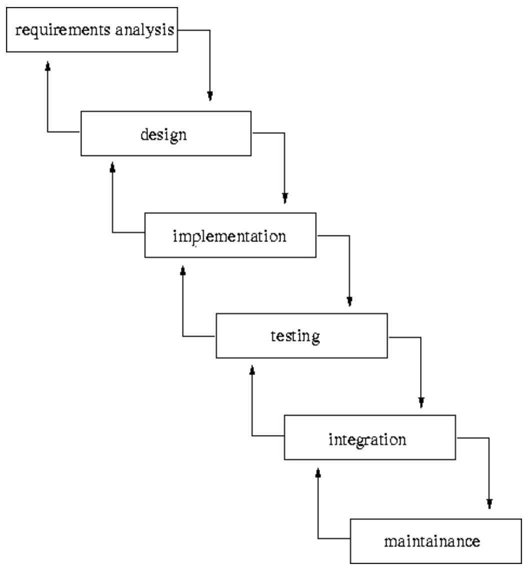

Figure 1.1: The waterfall model.

1.1.4 Call for Formal Methods

Although, industrial software must undergo rigorous testing phases before being shipped, the software errors mentioned in section 1.1.1 and 1.1.2 passed unnoticed, simply because their occurrence was unlikely. Thus, a simulation of the system will instead repeatedly visit the more likely configurations, while some less likely are never encountered.

As a reaction, over the last years research has spent considerable effort on formal methods, which allow to verify software in more systematic way than testing. The method of *model checking* has shown to be one of the most promising verification techniques until now. It is a highly generic verification scheme, which had successfully been applied to other areas such as process engineering and the verification of hardware. Based on a formal model of the investigated system, model checking uses exploration algorithms for a systematical enumeration of all reachable states. For each state, it is checked whether it fulfills certain properties.

There are several advantages of model checking software compared to testing . First of all, the exploration treats state transitions independent from their probability in real program execution. Thus, model checking can find subtle errors in programs which will most probably not arise during testing. Second, testing can merely provide information about the presence of errors but not about their cause. In contrast to testing, model checking builds a search tree of visited program states. Upon discovery of an error, the user is provided with the sequence of states (*trail*) that leads from the initial configuration to the error state. That information can be used to track down the error in the program much faster than given only the actual error state. As another advantage over testing, model checkers allow validating properties to be formulated using temporal logics such as LTL. That way, the user can precisely specify the expected behavior of the program.

Most current approaches for model checking programs rely on abstract models. These must either be constructed manually by the user or they are (semi-)automatically extracted from the source code of the inspected program. The use of abstract models can considerably reduce the memory requirements for model checking a program, but it also yields some drawbacks. Manual construction of the model is time consuming and error prone and may not correctly reflect the actual program. Also, the abstraction may hide exactly those aspects of the program which lead to errors in practice. Furthermore, the trail leading to some error also refers to the abstract model and must be mapped to the original program in order to find the source of the error.

As a recent trend, tools such as the Java model checker JPF [HP00] are capable of exploring the compiled program rather than an abstract model using a virtual machine. This resolves the drawbacks discussed above: Fist of all, the construction of a model is no longer required, which saves time. Furthermore, errors can no longer be abstracted-away, since the explored states represent the actual program configuration encoded in the processor registers and the computer's main memory. Also, the returned trail directly corresponds to positions in the programs source code and can be interpreted much easier than a sequence of states in the abstract model.

The single drawback of this approach lies in the high memory requirements of assembly-level program model checking. Due to the dynamic nature of actual programs, their state description can grow to an arbitrary size and storing the expanded states may quickly exceed the available memory of the underlying system. Thus research on unabstracted software model checking must mainly focus on reducing memory requirements.

1.2 History and Goals

The goal of this thesis is to give a clear vision of assembly-level software model checking. It introduces an experimental C++ model checker called StEAM (State Exploring Assembly Model Checker), which was developed in the course of the author's research.

StEAM can be used to detect errors in different phases of the software development process (cf. section 1.1.3). In the design and implementation phase, the engineers may build a prototype of the software, which reflects the system specification through a set of rudimentary C++ functions. This prototype can then be used to search for specification errors at the source code level. In later stages of development, more details are added to the prototype, while the system should be checked for errors at regular intervals. Along with the implementation progress, the type of errors that are searched for, gradually shift from specification to implementation errors.

As one advantage, this approach does not impose an additional overhead for constructing a formal model from the system specification in order to check properties in the design phase. Moreover, there is no need for the engineer to interpret error information from a formal model to the source code of the program - this is discussed more detailed in sections 2.4.1 and 2.4.2.

The tool will be used to exemplify the problems and challenges of unabstracted software model checking and possible solutions to them. As a clear focus, we concentrate on approaches that help to reduce the memory requirements. We will discuss methods for compact storing of states - such as collapse compression and bitstate hashing and how they apply in the context of C++ model checking. As another important issue we deal with methods that reduce the number of explored states, including the atomic execution of code blocks and the use of heuristics. Furthermore, we discuss time/space tradeoffs like state reconstruction, which accept a higher computational effort in exchange for lower memory requirements.

Each approach is theoretically discussed, implemented and empirically tested. In the end, the sum of the most effective techniques should allow us to apply unabstracted model checking on some simple but yet non-trivial programs.

Moreover, we discuss other tools used for model checking of software and point out the differences to our approach.

1.3 Structure

The thesis is structured as follows: In Chapter 2, we give an introduction to model checking and its application to software. Then, Chapter 3 introduces to general state space search and heuristics, which is essential for model checking. In Chapter 4, we give an introduction to the new assembly-level model checker StEAM, its capabilities and the supported heuristics. We also comment on how StEAM differs from current sate-of-the-art software model checkers. Then, in chapter 5 we point out the similarities between model checking and action planning. To emphasize the relation between the two fields, we use StEAM as a planner for concurrent multi-agent systems. Chapter 6 is dedicated to finding an efficient hashing scheme for StEAM, which turns out to be one of the core challenges in assembly-level software model checking. In Chapter 7, we discuss the method of *state reconstruction* as a way to alleviate the high memory requirements faced by tools that model check actual programs. In Chapter 8, we conduct experiments on heuristic search, hashing and state reconstruction. Finally, in Chapter 9 we summarize the contribution of the thesis and discuss future work.

The appendix contains the user manual of StEAM as well as technical details about the tool's implementation.

Chapter 2

Model Checking

2.1 Classical Model Checking

Model checking is a formal verification method for state based systems, which has been successfully applied in various fields, including process engineering, hardware design and protocol verification. In this section, we first clarify some terms that will be used throughout the thesis, some of which interchangeably as synonyms. Section 2.1.4 gives an example of an elevator system and checks it against some formal properties. It finishes by motivating the use of model checking and introducing to two of the most important classical model checkers.

2.1.1 Graph-Based Formalisms

Model checking generally refers to search in directed graphs. However, the literature does not obey to a standard formalism to denote a state space that is explored by a model checker. Some authors refer to or *Kripke-Structures* [LMS02], others to *(deterministic) finite automata* [AY98]. In some practically oriented papers, no formalism is given at all and the authors simply refer to *states* that are expanded [VHBP00b]. In fact, most graph-based formalisms used in model checking literature are equal in their expressiveness. We will exemplify this by giving a conversion between Kripke-Structures and deterministic finite automata.

Deterministic Finite Automata (DFA) are quadruples (S, I, A, T), where S is a set of states, $I \in S$ is the initial state, A a set of actions, and $T : S \times A \to S$ is a partial transition function mapping state/action pairs to successor states. In the theory of formal languages, a DFA additionally holds a set $E \subseteq S$ of accepting states. A DFA is said to accept a word $w \in A^*$, if starting at I, the corresponding sequence of actions leads to an accepting state. In model checking, automata are used to model the configuration space of a system, rather than to define formal languages, accepting states are usually not mentioned in definitions.

Kripke Structures (KS) are quintuples (S, I, Σ, T, L), where S is a set of states, I the initial state, $T \subseteq S \times S$ is a transition relation, Σ a set of propositions, and $L : S \rightarrow 2^{\Sigma}$ is a labeling function which assigns proposition sets to states.

KS ⇒ DFA Given a KS (S, I, Σ, T, L), we can construct an equivalent DFA (S', I', A, T') as follows: Let $S' = 2^{\Sigma}$, $I' = L(I)$ and $A = \{a_1, .., a_n\}$, where

$$n = max\{m | \exists s, s_1, .., s_m : (s, s_i) \in T, 1 \leq i \leq m, s_i \neq s_j \Rightarrow i \neq j\}$$

I.e., n is the maximal degree of outgoing transitions of a state in S. Furthermore, we define $T' = \bigcup_{(s,s') \in T}(s, a, s')$, such that for $(s, a, s'), (s'', a', s''') \in T', s \neq s'', s' \neq s''' \Rightarrow a \neq a'$.

The constructed DFA has a unique state for each set of labels which occurs in the KS.

DFA ⇒ KS Analogously, given a DFA (S, I, A, T), an equivalent KS (S', I', Σ, T', L') can be constructed. Let $\Sigma = A$, and $I' = I \in S'$. We regard a transition (s, a, s') as a directed edge from s to s' labeled with a. For each state $s \in S$ and each incoming edge with label a, we define a state $s_a \in S'$. Given S', we define $T' = \{(s_x, s'_y) | \exists (s, x, t), (s, y, s') \in T\}$.

Now, given a sequence of actions w the DFA will hold in state s, if and only if there is a path $I', s_1, .., s_n$ in the KS, such that $(L(s_1), .., L(s_n)) = w$.

In the rest of the thesis we may vary the formalism through which a state space is described, depending on what fits best in the current context.

2.1.2 Models, Systems, Programs, Protocols

Model checking is generally described as investigating the formal *model* of a system. This definition is somewhat misleading for new approaches like unabstracted software model checking (cf. Section 2.4), where the actual software is checked instead of a model.

Moreover, *protocols* control the behavior of a *system*. They restrict the state transitions that are allowed in the system (or in the corresponding formal model).

In practice, protocols are implemented as *programs* in a specific programming language. Just like protocols, these implementations can be model checked to detect errors.

In the rest of the thesis, the terms will be used as follows:

- Here, everything that can undergo a set of configurations is called a *system*. This includes physical systems like an elevator just like logical systems such as software.
- A *protocol* or *program* is a declarative description of legal transitions in a state space. For software, the states are implicitly defined by the values of the variables and the current position within the program. In other areas, like hardware verification, the state space is obtained by building a formal model of the underlying system.

2.1.3 Definition of Model Checking

Given a (state-based) formal model $\mathcal{M}$ and a property ϕ, model checking uses exploration algorithms to explicitly enumerate each state of $\mathcal{M}$. For each visited state s, it is checked, whether ϕ holds for s. A state s violating ϕ is called *error state*. If during exploration an error state is encountered, the model checker returns the sequence of states, the *error trail*, which leads from the initial to the error state. If the model $\mathcal{M}$ contains no error states, we say that $\mathcal{M}$ fulfills (models) ϕ.

Concurrency Many model checkers aim at the detection of errors in systems involving several concurrent processes. The false interleaving of process executions is a source for numerous and hard to find errors. Consider n processes, each yielding a set S_i of states. Then, the state space of the investigated system is formed by the cross product $S_1 \times .. \times S_n$ of all processes' state spaces. Note, that depending on the protocol or program which restricts the allowed state transitions, the cross product forms a superset of all reachable states.

Explicit vs Symbolic Model Checking Model checking approaches form two classes through the way they represent their explored states: In *explicit* model checking, each explored state is explicitly present in memory, while symbolic model checking uses declarative descriptions, e.g., logic formulae to represent sets of states. A symbolic model checker can, for example, use binary decision diagrams [McM92] for a compact representation of the logic formula that describes the set of explored states. Thus, it is possible to enumerate significantly larger states than with an explicit state representation [BCM+90]. Alternatives are bounded model checkers as presented in [BCCZ99]. As a clear disadvantage, symbolic model checking does not explicitly build a tree of visited states. Among other problems, this forbids the use of heuristics which rely on the structure of the search tree. Despite the potential of symbolic state representations, the work at hand will thus focus on explicit model checking, as the new software model checker presented here heavily relies on the use of heuristics to be evaluated on individual states.

2.1.4 Example

We use a simple example to illustrate the upcoming formal descriptions. Assume, that we want to design a control software for an elevator. The model involves three types of components (or processes). The first is the elevator, which can either be moving or at one of two floors. The second component is the elevator's door which can either be open or closed. The third component type is a button, of which the model has two instances (one for each floor). A button can either be idle or pressed. Figure 2.1 illustrates the three component types as deterministic finite automata.

The protocol, which controls the elevator's behavior, should fulfill several properties that guarantee the saftey and fair treatment of the passengers. These include, for instance, that the door is always closed when the elevator is moving and that each request for the elevator is eventually followed.

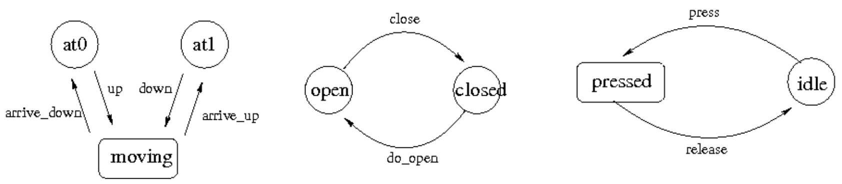

Figure 2.1: Three components of the elevator example.

According to Figure 2.1, the example may seem overly simplistic. We may start with a trivial protocol, that does not restrict the transitions of the system. Figure 2.2 shows the cross product of the automata - note the two instances for the button automaton. For clarity, we introduced an additional edge labeled with the action *start*. The edge has no predecessor and leads to the model's initial state $(at0, closed, idle, idle)$. The size of the product automaton illustrates, what is known as the *state explosion problem*. This combinatorial problem refers to the fact, that a system's state space grows exponentially in the number of its components.

2.1.5 Temporal Logics

Expressions in temporal logics allow logical formulae defined over the propositions of a labeled transition system S to be quantified over the paths of S. Temporal properties can often be integrated via a cross product of the automaton for the model and the automaton for the negated property. Examples for temporal logics are CTL [CES86] (Computation Tree Logic) and LTL (Linear Temporal Logic) [Pun77, BK00, ELLL01]. Temporal logics allow the basic logical operators $\vee, \wedge, \neg$ to describe state properties through basic terms over atomic propositions. By applying path quantifiers to basic terms, we arrive at temporal formulae, like:

$$AG(p \Rightarrow F\neg p)$$

The above formula *(always globally p implies future not p)*, asserts that along each path of the search tree, if at some state p holds, then there must be a future state at which p does not hold. We pass on a detailed description of temporal logics, as they are currently not used in the new software model checker presented in Chapter 4 and listing their formal syntax and semantics would be out of the thesis' scope. As temporal logics are an important aspect of classical model checking, the term should be mentioned though. Moreover, the remaining part of the elevator example given in this chapter uses temporal logic formulae to describe properties.

2.1.6 Types of Properties

Protocol verification distinguishes several classes of properties the system can be checked against. These properties are formulated either by *assertions* or temporal logic formulae. Assertions describe locally checked properties formulated as logical expressions over the associated values of a state. Depending on the underlying state model these values may correspond to the associated atomic propositions or to the content of system variables.

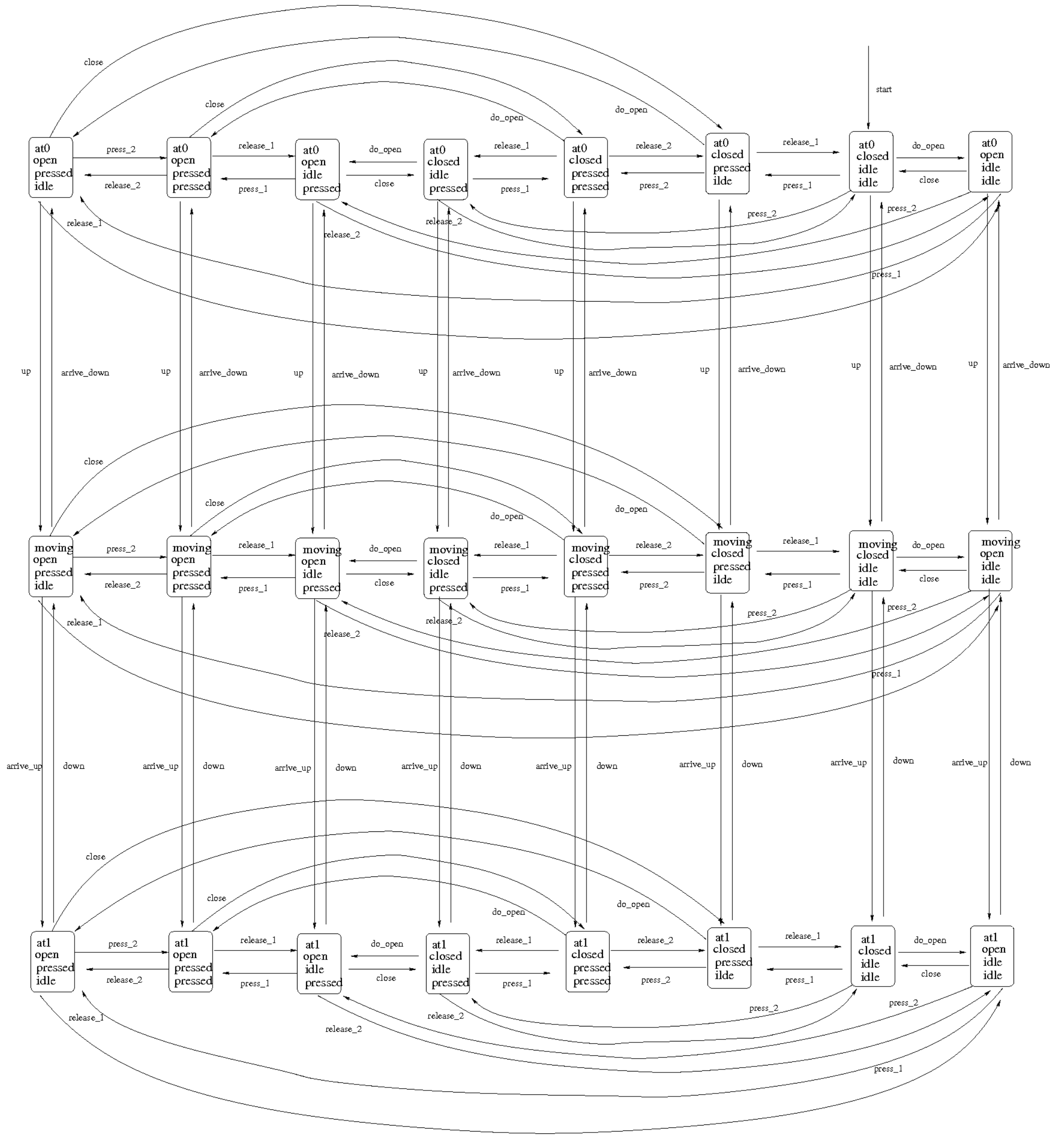
Figure 2.2: The state space of the elevator protocol.

Expressions in temporal logics quantify the aforesaid properties over paths in the state space of the system. Hence, temporal logics are more expressive than assertions, as the described properties can be checked independently from the position within the investigated protocol.

In the following, we discuss three important property classes in model checking.

Safety-Properties Safety properties describe states that should never be reached in the protocol. For our example we may consider that the elevator should never be moving while the door is open, i.e., $AG(\neg moving \vee closed)$.

Liveness-Properties Liveness-properties demand that a certain property will eventually hold in a state. For the elevator example, such a property might be that when a button was pressed at a floor, the elevator will eventually arrive at that floor in a future state: $A(pressed_i \Rightarrow F(at_i))$.

Absence of Deadlocks Deadlocks describe a systems state in which no progress is possible. This often occurs in systems involving concurrent processes, which request exclusive access to shared resources. There, a deadlock occurs if all processes wait for a resource to be released by another process. The elevator example is obviously deadlock-free. A example for a Deadlock is given in Section 8.1.

2.2 Towards a Fault-Free Protocol

The design of a fault-free protocol constitutes an evolutionary process. Starting with a prototype, it is checked if the protocol violates a desired property. If so, the error source is analysed and eliminated leading to a new protocol version. The cycle is repeated until no further violations are found.

It is not difficult to detect a property-violation in the initial protocol of the elevator example. The action sequence (or error trail) $start, open, up$ leads to an error state given by $(moving, open, idle, idle)$, which violates our safety property. An attempt to fix the error might be to inhibit the action *up* in states, where *open* holds. The smart system designer may also realize the analogous case, and forbid the action *down*, when the door is open. The resulting state space is depicted in Figure 2.3.

It differs from the initial state space in so far, that some transitions were removed. However, the new version still violates our safety property: The door may also open while the elevator is moving, i.e., $(start, up, open)$ is an error trail. As a consequence, we may forbid the *open* transition, while the elevator is moving. The resulting state space is depicted in Figure 2.4. For clarity, we have removed all unreachable states along with their in- and outgoing edges. As can be seen, there are no more states where both *moving* and *open* hold.

The system fulfills the safety property, however the new version of the protocol still violates our liveness property, since there exists an infinite sequence of actions:
$start, press1, release1, press1, release1, ...$ for which the systems cycles between two states at floor 0. Also, the elevator may simply open and close the door infinitely often. We try to fix this problem by adding, two more restrictions. First, when a button was pressed, the elevator may no longer open the door. Second, a button at floor i must immediately be released if and only if the elevator is at floor i. The resulting state space is depicted in Figure 2.5.

2.2.1 Need for Model Checking

In Section 2.2, we have checked a simple example against two properties. According to the violations we found, the protocol underwent several changes. In the example we were able to detected the violating action sequences manually, for three reasons. First,

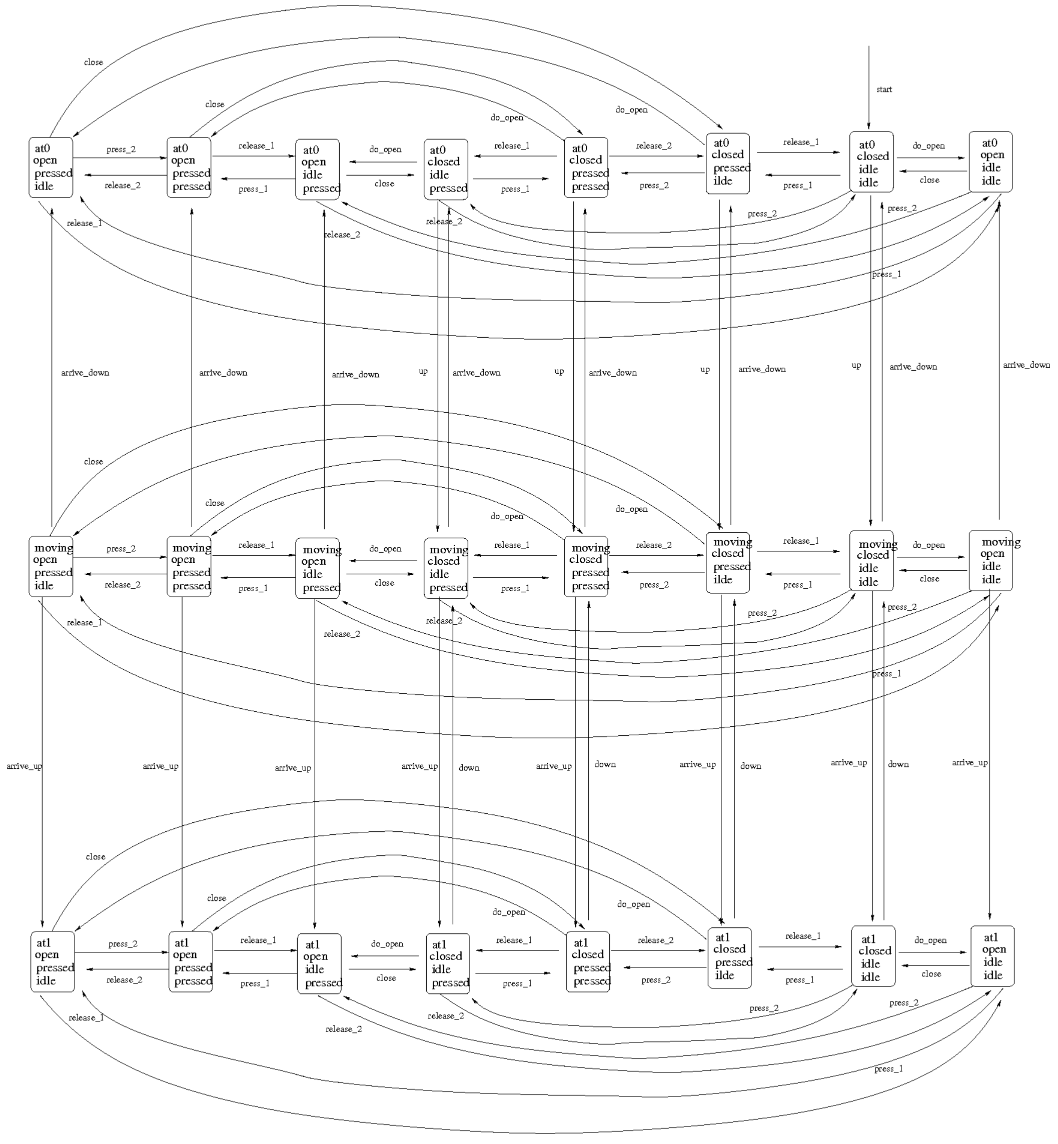

Figure 2.3: State space for second version of the elevator protocol.

the described system is very simple. Second, we started with the trivial protocol, that does not restrict the transitions and hence contains a lot of errors. Last, but not least, the example was designed for illustration purposes, and we a-priori knew the errors.

Limits of Code Inspection If we closely look at Figure 2.5, we realize that the system still contains errors. For instance, the elevator can be forced to remain at one floor if the corresponding button is repeatedly pressed. Such errors are already harder to detect by manual code inspection even in a system as simple as the elevator example, while for realistically sized examples the chance of finding an error converges towards zero.

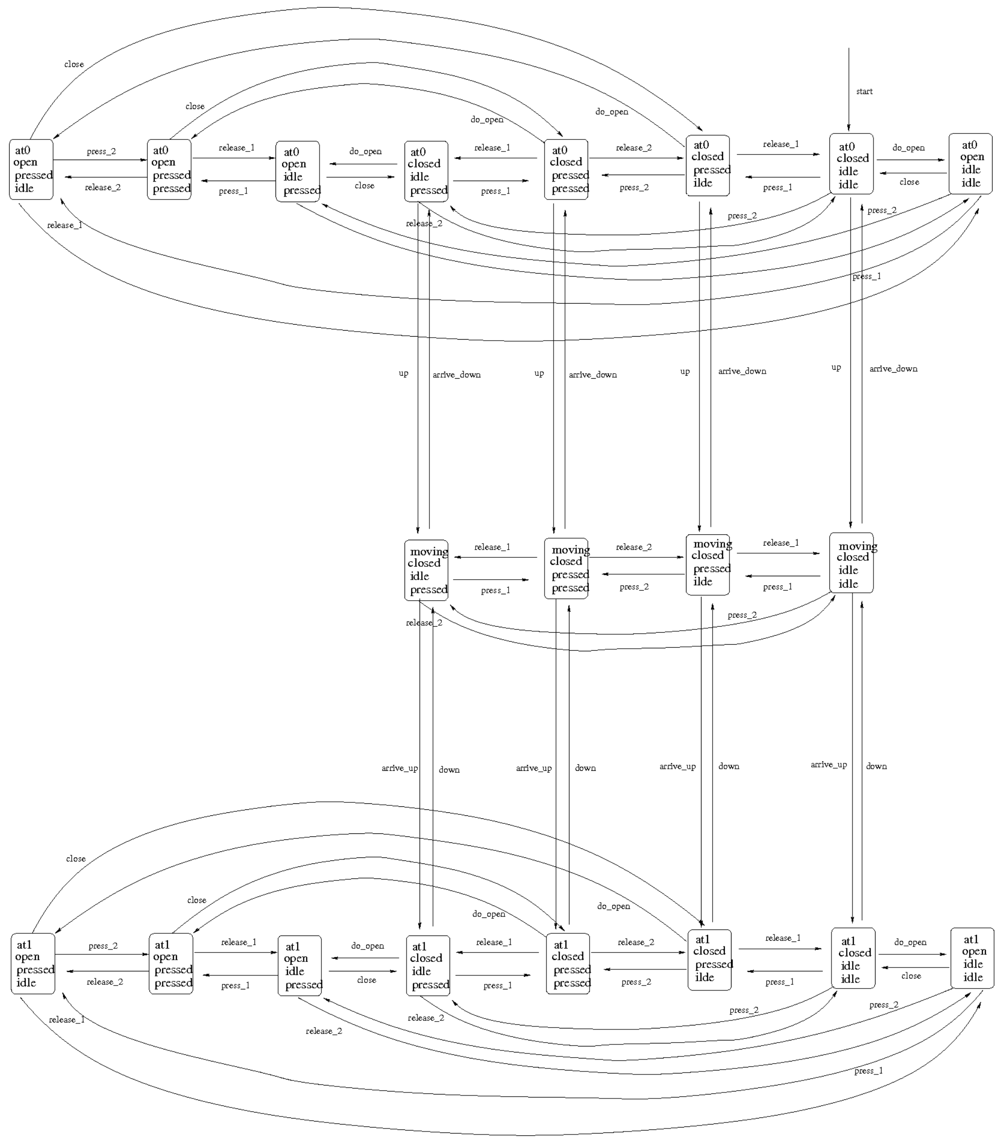

Figure 2.4: State space of the third version of the elevator protocol, fulfilling the safety property.

Limits of Testing The term *testing* refers to running a protocol or program in a simulated environment while checking if any unwanted behavior is exposed. Industrial standards demand quality criteria like a coverage of at least 99% of the code, before the program is deployed. However, such criteria only roughly correlate with the visited fraction of all reachable states, as states with equal code positions can differ in their variable configuration. Furthermore, state transitions occur with different probabilities. As a consequence, testing will visit certain states frequently, while other states are never reached. In contrast, model checking systematically enumerates all possible system configurations regardless of their probability. Thus model checking is able to detect subtle errors which pass unnoticed during testing. As a further advantage over testing,

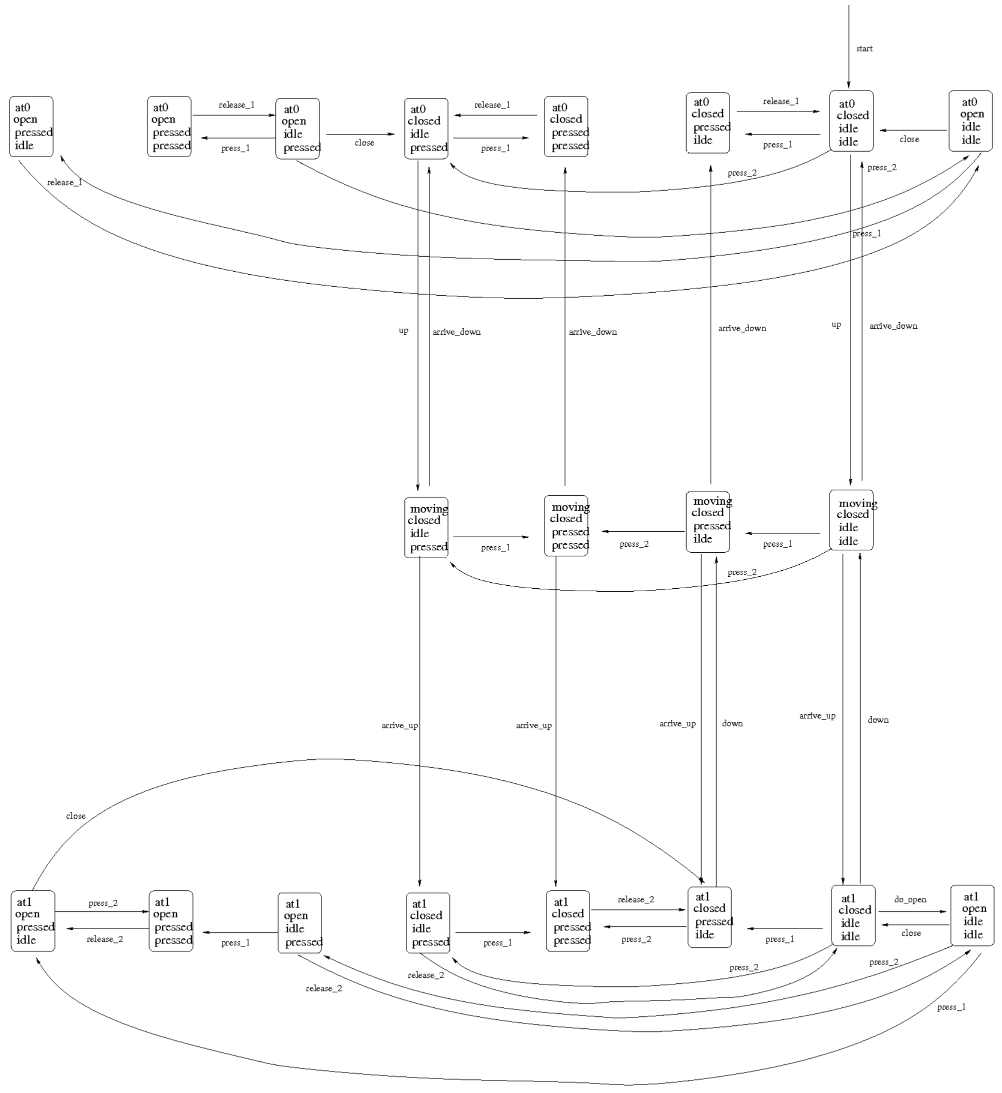

Figure 2.5: State space of the fourth version of the elevator example.

model checking builds a search tree of visited states, which makes it possible to return an error trail. This makes it easier for the user to detect the source of the error.

2.3 Classical Model Checkers

2.3.1 SPIN

SPIN [Hol97a, Hol03] is one of the oldest and most advanced model checkers. The tool uses the C-like description language *Promela* to describe a system with concurrent processes, whose state space can be explicitly enumerated. Due to its popularity, there are numerous derivations of SPIN like *dSPIN* [DIS99], which allows dynamic process creation or *HSF-spin* [ELL01], which uses heuristic search to speed up the search for errors. Also, some tools use SPIN as a back end to check Promela code generated through the conversion of program source codes [HP00]. SPIN is also the eponym for the annual workshop on model checking of software.

2.3.2 SMV

The tool SMV [McM92] developed in the Carnegie-Mellon university is a BDD-based symbolic model checker. Using a strongly declarative description language, models are described as networks of communicating automata. The properties under investigation are formulated as CTL formulae over the model. SMV owns its popularity to the fact, that it was the first model checker to use BDDs for symbolic exploration, enabling it to fully explore very large systems. Recent development based on SMV is *nuSMV* a reimplementation of the tool, which offers an open architecture that can be used as a core for custom verification tools [CCGR99, CEMCG^{+}02].

2.4 Model Checking Software

Recent applications of model checking technology deal with the verification of software implementations (rather than checking a formal specification). The advantage of this approach is manifold when compared to the modus operandi in the established software development cycle. For safety-critical software, the designers would normally write the system specification in a formal language like *Z* [Spi92] and (manually) prove formal properties over that specification. When the development process gradually shifts towards the implementation phase, the specification must be completely rewritten in the actual programming language (usually C++). On the one hand, this implies an additional overhead, as the same program logic is merely re-formulated in a different language. On the other hand, the re-writing is prone to errors and may falsify properties that held in the formal specification.

In contrast, when software model checking is used, the system specification can be formulated as a framework written directly in the targeted programming language. At that point, the framework can be checked for conceptual errors that violate formal properties. In the implementation phase, the framework is gradually augmented with implementation details, while the implementation is regularly checked against formal properties in the same fashion as done in the specification phase. In contrast, violations can now be attributed to implementation errors, rather than conceptional errors in the specification - given that we check against the same properties.

Moreover, the systematic verification of model checking is superior to any manual investigation of the formal specification: it is less likely for undetected conceptual errors to carry over to the implementation phase, where they are much more difficult to fix. Model checking is also superior to manual inspection or testing of the implementation, as the systematic exploration will not miss states that are unlikely to occur in the actual execution of the program, which means that less errors are carried over from the implementation- to the integration phase.

2.4.1 The Classical Approach

In the classical approach of software model checking, an abstract model of the program is investigated. The model must either be constructed manually or (semi-)automatically generated from the program's source code. The model can then be investigated by established model checkers like SPIN or SMV.

As an advantage of the verification of abstract models, the user can use the full range of features of highly developed model checkers. The main disadvantage lies in the discrepancy between the actual program and its abstract model. Fist of all, the model might abstract way details, which lead to an error in practice. Second, the model can lead to *false positives* - i.e. errors that are present in the model but not in the actual program. This may just invalidate the advantages of model checking over testing, as we cannot rely on the states - though systematically enumerated - to be representative for the actual program.

The automatic translation of source code by custom-build parsers also yields problems: First, modern programming languages such C++ have complex formal semantics, which implies that devising a parser for the automatic conversion is tedious and time-consuming. The more time is spent for that parser, the less time remains for the design and implementation of sophisticated model checking technology. As a workaround, many tools only support a subsets of the programming language's formal semantics.

As another disadvantage, the returned error trail refers to the model and not to the source code of the inspected program, which makes it harder to track down and fix the error in the actual program source code.

Moreover, as a very severe problem, the structure and dynamics of computer programs, such as memory references, dynamic memory allocation, recursive functions, function pointers and garbage collection can often not satisfactorily be modeled by the input languages of classical model checkers.

2.4.2 The Modern Approach

The modern approach to software model checking relies on the extension or implementation of architectures capable of interpreting machine code. These architectures include virtual machines [VHBP00a, ML03] and debuggers [MJ05].

Such unabstracted software model checking does not suffer from any of the problems of the classic approach. Neither the user is burdened with the task of building an error-prone model of the program, nor there is a need to develop a parser that translates (subsets of) the targeted programming language into the language of the model checker. Instead, any established compiler for the respective programming language can be used (e.g. GCC for C++).

Given that the underlying infrastructure (virtual machine) works correctly, we can assume that the model checker is capable of detecting all errors and that it will only report real errors. Also, the model checker can provide the user with an error trail on the source level. Not only does this facilitate to detect the error in the actual program, the user is also not required to be familiar with the specialized modeling languages, such as Promela.

As its main disadvantage, unabstracted software model checking may expose a large state description, since a state must memorize the contents of the stack and all allocated memory regions. As a consequence, the generated states may quickly exceed the computer's available memory. Also, as will be seen later, a larger state description will slow down the exploration. Therefore, one important topic in the development of an unabstracted software model checker is to devise techniques that can handle the potentially large states.

2.4.3 Some Software Model Checkers

dSPIN

The tool dSPIN is not primarily a software model checker, but rather a dynamic extension to SPIN. The motivation for the tool came from previous experiences regarding attempts to convert Java programs to Promela [IDS98]. The extensions relate to dynamic features of programs, which should allow the modeling of object-oriented programs in a natural manner. The extensions include pointers, left-value operators, new/delete-statements, function definition/call, function pointers and local scopes [DIS99].

The tool alleviates the problem of modeling actual computer programs. However, it still imposes many restrictions to what language constructs can be modeled. For instance, the ampersand operator cannot be applied to pointer variables, which implies that multi-level pointers cannot be fully handled in the language of dSPIN. Also, the generated counter examples are based on the modeling language and must be manually re-interpreted to the actual program source.

Bandera

The Bandera [HT99] project of the Kansas State University is a multi functional tool for model checking Java programs. It 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. Also, Bandera serves as a frontend for JPF2 (cf. Section 2.4.3). The approach is based on a custom-build intermediate language called BIR (Bandera Intermediate Language). The advantage is, that Bandera does not need to implement the model checking technology itself, but rather inherits it from the back-end tools. However, in consequence it also suffers from the same drawbacks as the other abstraction-based model checkers. Even if the intermediate language allows an adequate modeling of Java programs, it still needs to be converted to the input language of the back-end model checker which brings up the problems of handling dynamic issues of real programs. Moreover, since Bandera supports a couple of back-end tools, it should be difficult to maintain, as it needs to catch up with the features new versions, such as new search algorithms and heuristics.

SPIN 4.0

In the versions 4.0 and above, Spin allows embedding of C code in the Promela models. Yet, this does not enable Spin to model check programs on the source code level. The embedded code is blindly copied into the source of the generated verifier and cannot be checked [Hol03]. The idea is to provide a way to define guards, states, transitions and data types in an external language within the model.

CMC

CMC [MPC$^+$02], the ”C Model Checker”, checks C and C++ implementations directly by generating the state space of the analyzed system during execution. CMC is mainly used to check correctness properties of network protocols. The checked correctness properties are assertion violations, global invariant checks avoiding routing loops, sanity checks on table entries and messages as well as memory leaks. The approach of CMC is based on extending the checked program with a testing environment which executes single steps of the (multi-threaded) program while checking for correctness-properties in-between. This implies, that CMC does not possess a meta-level infrastructure (e.g. a virtual machine) which has full control over the program execution. In particular, the resulting state of a transition is only available, *after* the corresponding machine code was physically executed on the computer on which the model checker runs. This makes it impossible to detect fatal program errors, such as illegal memory accesses, since these lead to a crash of the executed program and thus of the model checker. In contrast, tools based on machine-code interpreting infrastructure can check the validity of each machine instruction, before it is executed.

VeriSoft

The tool VeriSoft [God97] is a program model checker which can be applied to check any software, independently from the underlying programming language. VeriSoft offers two modes: interactive simulation and automatic verification. In the latter, the tool systematically enumerates the state space of the investigated program. The supported error types are deadlocks, livelocks, divergences, and assertion violations. A livelock means, that a process has no enabled transitions for a specifies amount of time. Divergences occur, when a process does not execute any visible operation during a user-specified time. When an error is found, the respective counter example is presented to the user in a interactive graphical interface. Like CMC, the approach is based on a supervised program execution by augmenting the source code of the verified program with additional code. This a priori excludes the possibility to find low-level errors such as illegal memory accesses. Moreover, VeriSoft explores programs based on input- output behavior rather than on the source level of the investigated program.

Zing

The Zing [AQR$^+$04a, AQR$^+$04b] model checker developed by Microsoft Research aims at finding bugs in software at various levels such as high-level protocol descriptions, workflow specifications, web services, device drivers, and protocols in the core of the operating system. Zing provides its own modeling language and an infrastructure for generating models directly from software source code. In a recent case study[1], it was possible to find a concurrency error in a Windows device driver. This indicates, that Zing may have a practical relevance for checking properties of actual programs. Still, the approach faces the same problems like all software model checkers based on formal models, i.e. the model may abstract-away details of the program which lead to an error in practice or it may introduce false positives.

SLAM

The SLAM project by Microsoft research is a model checker for sequential C programs, based on static analysis. It uses the specification language SLIC to define safety properties over the program. Given a program P and a SLIC specification, an instrumental program P' is created, such that a label ERROR in P' is reachable, if and only if P does not satisfy the specification [BR02]. The process in SLAM generates a sound Boolean abstraction B' from the instrumental program P', where sound means, that if ERROR is reachable in P', then it is also reachable in B'. However, the reverse cannot be assumed, i.e. the inspection of B' may yield false positives. To deal with the latter, the tool uses *counter-example driven refinement*, which generates a more detailed abstraction B'' of P', containing less spurious paths than B'. The approach of SLAM is very interesting, as the tool is based on abstract models, while it adresses one of its main problems - i.e. false positives. Yet, the tool works on a function call/return level and is mainly targeted to programs that implement an underlying protocol's state machine. Thus, it is presumably not possible to use this approach to find low-level errors such as illegal memory accesses.

BLAST

The *Berkley Lazy Abstraction Software verification Toolkit* (BLAST) [HJMS03] is a tool for the verification of safety properties in C programs. The verification scheme of BLAST uses control flow automata to represent C programs and works in two phases: in the first phase it constructs a reachability graph of reachable abstract states of the program. If an abstract error state is found, the second phase begins. Here, it is checked through symbolic execution of the program, whether the error is real or results from a too coarse abstraction of the program. Similar to SLAM, BLAST considers error states found in the abstract state space to be false positives and performs an additional analysis to test their genuity.

As a limitation, the current version of BLAST cannot accurately interpret C programs that involve function pointers or recursive function calls [Ber04].

[1]http://research.microsoft.com/zing/illustration.htm

Java PathFinder 1+2

The Java PathFinder [Hav99] (JPF) developed by the NASA is a model checker for concurrent Java programs. The name of the tool originates from the Mars rover robot *Mars Pathfinder*, which shut down due to an error in the control software [HP00]. The error had cost the NASA a considerable amount of money - money that could have been saved, if the software had undergone a more thorough investigation. This famous event raised an increased awareness to the need for automated software verification. In its first versions (JPF1), the tool was merely a converter from Java source code to Promela. Since JPF1 is no longer under development, the term 'JPF' always refers to the second version of the model checker: JPF(2) [VHBP00a], uses a custom-made Java Virtual Machine to perform model checking on the compiled Java Byte-Code. By this approach, 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 [GV02]. An interface allows the implementation of user defined heuristics (see e.g. [EM03]). Its powerful features and the public availability [2] has earned JPF a lot of attention in the model checking community. The unabstracted model checking approach, which eliminates various problems of previous technologies has also inspired other projects [ML03, MJ05] to use machine code interpreting infrastructure to model check actual programs.

Yet, JPF is limited to the verification of Java programs, while most software products - including that of the NASA - are written in the industrial standard programming language C++. In fact, the developers of JPF state that for the verification with JPF, some software was manually converted from C++ to Java [VHBP00b]. Not only does this imply and additional overhead, but may also invalidate some advantages of the JPF approach, since the manual conversion may induce errors, resulting in false positives and hiding actual software faults.

Estes

The Estes tool implements a quite recent [MJ05] approach to program model checking, which builds on the GNU debugger (GDB) to verify software on the assembly level. The approach differs from assembly model checkers such as JPF or StEAM in so far, as GDB supports multiple processor models. Hence the tool is not limited to the verification of high-level program properties. Instead, a stronger focus is laid on checking time critical properties of the compiled code of embedded software. In [MJ05] an example for an error is given that is not visible on the source level, but rather occurs due to an incorrect timing in the m68hc11 machine code. Here, the program reads two sensor values from registers that are updated by an interrupt which is invoked every x CPU cycles. A data inconsistency can occur, when the interrupt fires after the program has read the first register and before comparing it with the second.

Building on a debugger is an alternative approach, which yields the same advantages as enhancing virtual machines. Also, the possibility to search for errors that occur only on the machine code of a specific architecture make this a promising tool. Unfortunately,

[2] http://javapathfinder.sourceforge.net/

the currently available literature gives little technical detail about the implementation (e.g., how states are encoded), nor does it provide enough experimental data to really judge the capabilities and limits of this approach.

Summary

Besides state explosion, an inherent problem that most existing program model checkers face is the complexity of the formal semantics (FS) of modern programming languages such as C++. The more effort tools invest to account for details of the underlying FS, the less time remains for the development of sophisticated model checking techniques, such as exploration algorithms, compact state memorization and heuristics.

As one workaround, tools such as Bandera or BLAST support only a subset of the respective programming language's FS. Obviously, this burdens the user with the task of rewriting the application to a version, which is checkable by the respective tool. Not only does this impose an additional overhead to the software development process, but rather the rewriting process may introduce or hide errors of the actual program.

In a second workaround that is taken by e.g. CMC, the investigated programs source code is augmented with a model checking unit and compiled to an executable, which performs a stepwise, supervised execution of the program, while checking for errors between each step. This method is mainly suitable for checking highlevel properties such as safety-violations or deadlocks, rather than machine-level errors such as illegal memory writes. The problem is, that the machine instructions which correspond to a state transition of the program are executed on the actual hardware of the machine that runs the model checker. Since the result of a transition is only known, after the respective machine instructions were executed, fatal errors such as illegal memory writes cannot be caught, as they terminate the execution of the model checker.

The second version of the Java PathFinder remedies the above problems by using a virtual machine to execute the Byte Code of the investigated program. This ensures, that the model checker accounts for the complete FS of the underlying programming language, while there is no need to develop a translator from the program source code to some formal model - instead one can use any 3rd party Java compiler, that produces valid Byte Code. Obviously, the development of a virtual machine also imposes a considerable one-time overhead. In particular, it must be assured that the virtual machine works correctly, as errors in the machine may again lead to false positives even if the underlying formal model (the machine code) is correct. Model checking the virtual machine's implementation may help here, but that requires another virtual machine that works correctly.

Alternatively, a tool may use an existing virtual machine that was designed to run applications rather than for the use in a model checker exclusively. Such an infrastructure can be assumed to be thoroughly tested for correct functioning. However, it is always difficult to integrate external components into your own tool. As previously mentioned, the developers of JPF assumed the design of a custom-made virtual machine to be more feasible than tailoring a model checking infrastructure to an existing machine (such as the official Java interpreter).

As will be seen in Chapter 4, the C++ model checker StEAM takes the challenge of using a 3rd-party virtual machine to execute the machine instructions corresponding to state transitions in the investigated program. It will turn out, that the amount of work needed to do this is not more than that of developing a custom virtual machine - probably less. As the core contribution of the thesis, StEAM is the first model checker capable of checking concurrent C++ programs without the need to build formal models. This approach allows the investigation of programs written in the industrial standard programming language, while

- There is no need of manual conversion or heavy rewriting of the original program.
- The semantics of the program are accurately interpreted.
- The designers of the model checker are not required to write a compiler or virtual machine themselves.
- The errors that can be searched for are not restricted to violations of high-level properties, such as safety-violations or deadlocks. Additionally, the model checker can detect low-level errors, like illegal memory accesses.

Furthermore, the model checker StEAM constitutes a versatile tool whose field of application is not restricted to the verification of formal properties. Chapter 5 presents a technique to use the model checker as a planner for concurrent multi-agent system.

Chapter 3

State Space Search

Explicit model checking essentially relies on the exploration of state spaces with search algorithms. Here, we give an introduction to state space search and discuss the most important search algorithms and the advantages and drawbacks they impose in the field of (software-) model checking.

3.1 General State Expansion Algorithm

A general search algorithm systematically enumerates the nodes of a *directed graph*. A directed graph is a tuple (V, E), where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. Graph nodes are often used to model the states of a system, hence the terms *node* and *state* may be used interchangeably. During the enumeration of the nodes, a search algorithm maintains two lists: *open* and *closed*. The *closed*-list contains all fully expanded nodes. A node n is declared to be fully expanded , if all immediate successor states $\{n'|(n, n') \in E\}$ were visited. The list *open*, contains all nodes of the explored graph that have been visited, but not yet fully expanded. Initially, the *closed* list is empty and *open* only contains some designated start node $s \in V$. In each step, the algorithm removes a node n from $open$ according to the used search strategy. Then, n is expanded - i.e. each immediate successor node n' with $(n, n') \in E$ is visited and added to the $open$ list, if n' is not already in the *closed* or *open* list. After state n has been expanded, it is added to the closed list. The algorithm terminates, when the $open$ list is empty, which implies that all reachable nodes were visited. Algorithm 3.1 shows the pseudo code for the general state expansion algorithm. Here, the function $expand(n)$ returns all immediate neighbors of node n - i.e. $expand(n) = \{n'|(n, n') \in E\}$.

Common search algorithms are basically implementations of the abstract description in algorithm 3.1 and differ mainly in the criterion upon which the next open state is selected for expansion.

The working method of different search algorithms can very naturally be exemplified for geometric search. Here, the costs of a path from a start location to the goal corresponds to the sum of covered distances between the intermediate locations in that path. In the graph depicted in figure 3.1, we have a start node i, goal node g and seven other nodes $s_1, \ldots, s_7$. For simplicity, we assume, that the spacial arrangement of the nodes in

```
**Procedure** GSEA(V,E,$\gamma$)

Input: A directed graph (V,E), a start node $i$ and a goal description $\gamma$.
Output: Goal state $g$ or $null$.

$open = \{i\}$ ;$closed = \emptyset$
while ($open \neq \emptyset$)
    select $n \in open$
    if ($n \models \gamma$) return n
    $open \leftarrow open \setminus \{n\}$
    $\Gamma \leftarrow expand(n)$
    $closed \leftarrow closed \cup \{n\}$
    $open \leftarrow open \cup (\Gamma \setminus closed)$
return null
```

Algorithm 3.1: General State Expansion Algorithm

Figure 3.1 is such, that the geometric distance between two locations is proportional to the Euclidean distance between their corresponding nodes. The optimal path from i to g obviously leads through s_2 and s_5.

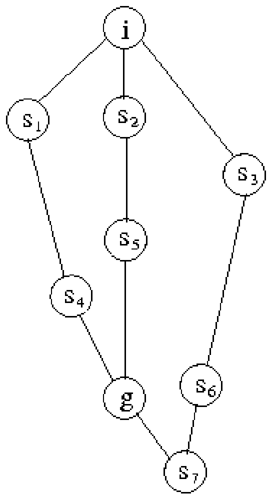

Figure 3.1: Example Network

3.2 Undirected Search

Undirected (aka blind) search algorithms have no information about the distance of a state to a goal state in the search space and instead select the open state to be expanded next according to its position in the search tree. The two most important undirected search algorithms are breadth-first and depth-first search.

3.2.1 Breadth-First Search

BFS expands in each step a node n with minimal depth (number of nodes in the path from i to n). This can be achieved by implementing the open list as a FIFO queue. BFS gives optimal paths in uniformly weighted graphs - i.e. in graphs where all transitions from some node to an immediate successor imply the same costs. Obviously, this is not the case for the example network in figure 3.2, since the costs correspond to the Euclidean distance between the nodes, which may be different for each pair of adjacent nodes. Hence, in geometric search, paths obtained by BFS may be suboptimal as shown in Figure 3.2, which depicts the search tree for the example network. Here we assume that of two nodes $s_i \neq s_j$ with the same depth and $i < j$, s_i is expanded first. The roman numbers indicate the order in which nodes are expanded. BFS requires five expansions and returns the path i, s_1, s_4, g.

BFS in Software Model Checking

Optimality of BFS in model checking depends on the subjective definition of the "costs" of counter examples (trails). These costs should be proportional to how difficult it is to track down the trail and use the information for fixing the error in the protocol or program. The assumption in this thesis is, that shorter trails can always more easily be interpreted by the user than longer trails. This implies that we give uniform costs for all state transitions, which makes BFS an optimal search algorithm in the domain of software model checking. However, since the size of the open list grows exponentially with the search depth, the space required to memorize the open states will often exceed the available memory, before an error is found.

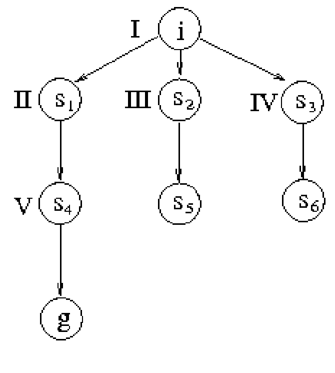

Figure 3.2: Path found by BFS

Note that there is more recent work on helping the user in diagnosing the cause of an error [GV03]. This may yield even better criteria concerning the quality of counter examples, but it also imposes a harder problem.

3.2.2 Depth-First Search

As the dual of BFS, depth-first search (DFS) selects the open node with maximal depth to be expanded next, which is achieved by implementing the open list as a stack. Obviously, DFS is not optimal as it tends to return the longest path from the start node to a goal node, which is most likely suboptimal even in non-uniformly weighted graphs.

Figure 3.3 shows the search tree for the example network as generated by DFS. Here, the path from the start- to the goal node is of length five - as opposed by four in BFS. On the other hand, DFS only requires four node expansions - in contrast to five expansions for BFS.

DFS in Software Model Checking

Depth-first search is an option in software model checking, if optimal algorithms fail to find an error. The advantage of DFS over BFS is that the number of open states grows only linearly with the encountered search depth. As its main drawback, the trails returned by DFS may be very long which makes it difficult or impossible for user to track down the error with the help of that trail. Moreover, DFS is incomplete in infinite search spaces that programs often expose.

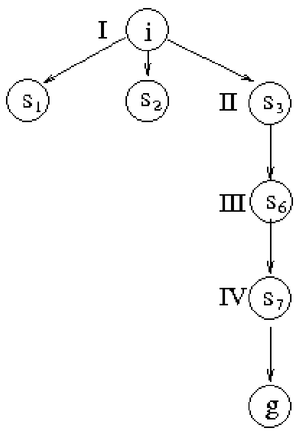

Figure 3.3: Path found by DFS

3.2.3 Depth-First Iterative Deepening

Given a depth limit d, Depth-first iterative deepening search (DFID), explores the state space in the same order as DFS, ignoring states with a depth greater than d. If no goal state is found, the search is re-initiated with an increased depth limit. DFID incorporates the low memory requirements of DFS with the optimality of BFS in uniformly weighted graphs. Moreover, DFID remedies the incompleteness of DFS in infinite state spaces. The drawback of DFID lies in the high time requirements, as each iteration must repeat all node expansions of the previous one.

DFID in Software Model Checking

DFID is very interesting for software model checking, as it constitutes an optimality-preserving time- space tradeoff. It is a known fact that due to the state-explosion problem, the failure of search algorithms can generally be attributed to a lack of space rather than a timeout. For the memory saving to take effect however, closed states may not be memorized explicitly. This requires the search to be integrated with compacting hash functions such as bitstate-hashing (cf. 6.2.1).

3.3 Heuristic Search

3.3.1 Best-first Search

Best-first search belongs to the category of *heuristic* search algorithms. The state to be expanded next is chosen as the state s which minimizes the estimated distance $h(s)$ to the goal (h is called the *estimator function* or simply the *heuristic*). An obvious estimator function for geometric searches is the Euclidean distance to the goal point i. Assuming an adequate heuristic, best-first search will in general find the goal state faster than blind search algorithms such as BFS. However, the obtained path is not guaranteed to be optimal. When applied to our example network using Euclidean distance as the heuristic estimate, the generated search tree is the same as for DFS (Figure 3.3). The preferability of heuristic search over undirected search critically depends on the quality of the used heuristic. A bad heuristic will not only return suboptimal paths, it will also lead to an increased number of node expansions as compared to e.g. DFS. Even though, the Euclidean distance is generally a good heuristic in geometric search, it may be misleading as the example shows. However, we may generally expect it to perform good in geometric networks.

Best-First Search in Software Model Checking

Best-first search is an important option in software model checking, when optimal search algorithms fail due to time- or space restrictions. With a properly chosen heuristic it is often possible to find an error. Experimental results (see e.g. chapter 8) show that with an adequate heuristic errors can be found with low time- and memory requirements while the generated error trails are close to optimal.

3.3.2 A*

The heuristic search algorithm A* is a refinement of best-first search. Here, the state s is expanded next, which minimizes $f(s) = h(s)+g(s)$, where h is a heuristic estimate and g the path cost from the initial state to s. In geometric search, the path costs correspond to the accumulated point distances, i.e. if $s_1, \dots, s_n$ is the path leading from initial state $i = s_1$ to state $s = s_n$, and $d(s', s'')$ denotes the Euclidean distance between point s' and point s'', then $g(s) = \sum_{i=1}^{n-1} d(s_i, s_{i+1})$

Admissibility and Consistency

A heuristic h is *admissible*, if the estimated goal distance $h(s)$ of some state s is always less or equal than the actual distance from s to g. A* provably returns the optimal path, if the used heuristic is admissible. The term admissible is also used to describe strategies that produce optimal results for a given class of problems - such as BFS in uniformly weighted graphs.

As a stronger property, a heuristic is *consistent*, if for each state s and each immediate successor s' of s it holds, that $h(s) \leq h(s') + c(s, s')$. Here, $c(s, s')$ denote the path costs from s to s'. When A* is used with a consistent heuristic, the f-value along each path is non-decreasing.

If we use A* combined with the Euclidean distance on a network of waypoints, we can expect that the returned path is optimal. Figure 3.4 shows the search tree we get by applying A* on the example network. Here, only three expansions are needed to obtain the optimal path i, s_2, s_5, g.

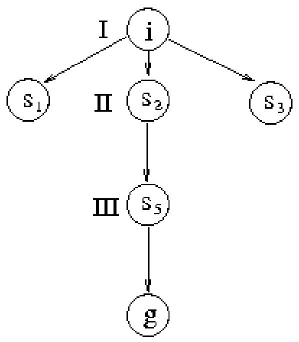

Figure 3.4: Path found by A*

A* in Software Model Checking

A severe problem of using A* in software model checking is, that most of the heuristics used are not admissible - one exception is the *most-blocked* heuristic used to detect deadlocks (cf. 4.5.1). Non-admissible heuristics are generally known to perform poorly under A* and the experimental results in chapter 8 support this claim. Still, A* is useful in software model checking e.g. if a suboptimal trail is to be improved. In this case, an admissible heuristic can be devised by abstracting the state description to a subset of its components.

3.3.3 IDA*

Like DFID (cf 3.2.3), Iterative Deepening A* (IDA*) [Kor85] performs a series of depth-first traversals of the search tree. In contrast to DFID, IDA* limits the $f - value$ $f(s) = h(s) + g(s)$ of the generated states, rather than their depth. Starting with an initial cost-bound $c = f(i)$, where i is the initial state, IDA* preforms a depth-first traversal while generated states with an f-value greater than c are not inserted to the open list. When no more open states remain, the new value of c is the minimal cost-value of all encountered states that exceeded the old value of c. Like A*, IDA* returns the optimal path form i to a goal state, if an admissible heuristic is used.

IDA* in Software Model Checking

Like DFID, IDA* is interesting for software model checking at points when other optimal algorithms fail due to lack of memory. Like A*, its applicability depends on whether an admissible heuristic is available.

Recent Development

There is a vast body of recent development of search algorithms not covered by the thesis. The algorithms presented in this chapter are the fundamental and most important ones. Moreover, with exception of IDA*, the presented algorithms are implemented in the new model checker StEAM presented in Chapter 4.

Chapter 4

StEAM

4.1 The Model Checker StEAM

StEAM is an experimental model checker for concurrent C++ programs. Like JPF, the tool builds on a virtual machine, but also takes some further challenges. To the best of the author's knowledge, StEAM is the fist C++ model checker, which does not rely on abstract models. An increased difficulty of C++ lies in the handling of untyped memory as opposed by Java, where all allocated memory can be attributed to instances of certain object types. Also, multi-threading in Java is covered by the class *Thread* from the standard SDK. Such a standardized interface for multi-threading does not exist in C++.

Moreover, StEAM build on an existing virtual machine, IVM. This is novel since tailoring a model checking engine to an existing virtual machine imposes a task, that was thought to be infeasible by the creators of JPF [VHBP00b]. IVM is further explained in section 4.2.

4.2 Architecture of the Internet C Virtual Machine

The *Internet Virtual Machine* (IVM) 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 is to be able to receive pre-compiled programs through the Internet and run them on an arbitrary platform without re-compilation. 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[1]. This is a strong empirical evidence that the virtual machine correctly reflects the formal semantics of C++. IVM is publicly available as open source[2].

[1]`www.doomworld.com/classicdoom/ports`
[2]`ivm.sourceforge.net`

The Compiler The compiler takes conventional C++ code and translates it into the machine code of the virtual machine. IVM 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 files 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, the symbol table etc., not to be considered for memorizing the state description.

State of IVM At each time, IVM is in a state corresponding to the program it executes. The structure of such a state is illustrated in Figure 4.1.

The memory containing an IVM state is divided in two hierarchical layers. One is the *program memory* containing the memory allocated by the simulated program. Program memory forms a subset of the other layer, the *physical memory* of the computer the virtual machine is running at. The physical memory contains the contents of CPU-registers (machine) and the *stack* of the running program. The CPU registers hold the stack and frame pointer (SP,FP), the program counter (PC) as well as several registers for integer and floating point arithmetic (Ri,Fi). The physical memory also holds the object file image (MI), which is essentially a copy of the compiled ELF-binary. The MI stores among other program information the machine code and the Data- and BSS-sections. Note, that the space for storing values of initialized global variables resides directly in the MI, while for uninitialized variables an additional block of memory is allocated at program startup.

In its original implementation, IVM does not provide a data structure which encapsulates its state, as it is implicitly described by the contents of the memory areas depicted in figure 4.1 and changes in a deterministic fashion. A fist goal in the development of StEAM is to formulate a state description as a data structure, which captures all relevant information about a program's state.

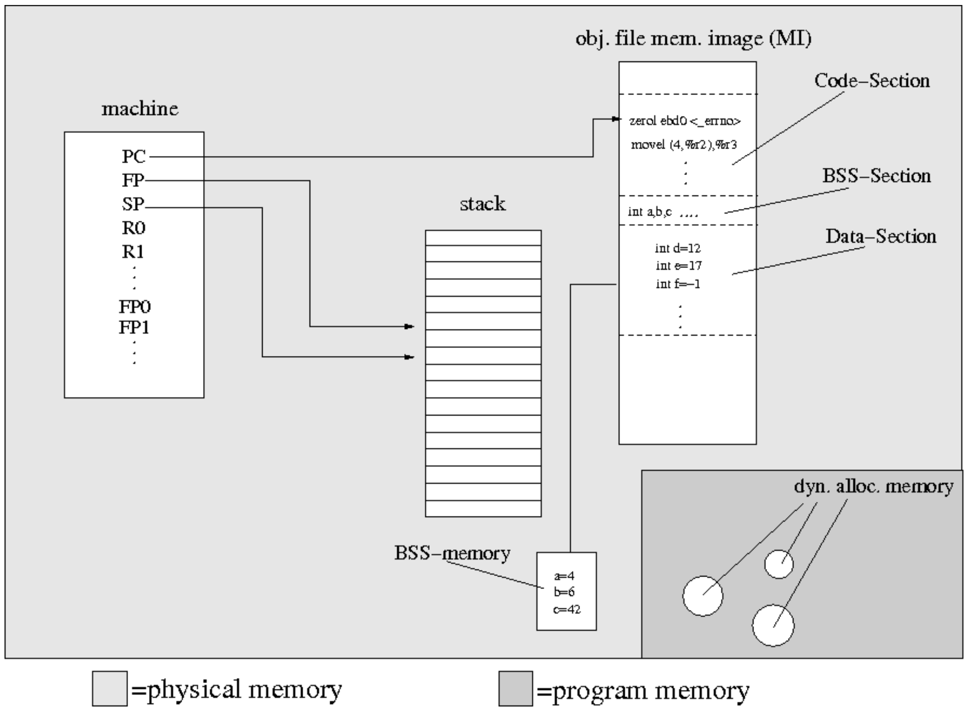

Figure 4.1: State of IVM.

4.3 Enhancements to IVM

In the following, we give a technical description of the enhancements used to tailor a model checking functionality to IVM. These include a state description, a multi-threading capability and a set of search algorithms that enumerate the program's state based on our description.

4.3.1 States in StEAM

As stated in Section 4.2, the fist step is to encapsulate the state description in a data structure, which contains all IVM components from Figure 4.1 as well as additional information for the model checker. The latter includes:

Threads: As model checking is particularly interesting for the verification of concurrent programs, the description should include all relevant information about an arbitrary number of running processes (threads).

Locks: Many concurrency errors involve deadlocks or the violation of access rights. Threads may claim and release exclusive access to a resource by *locking* and *unlocking* it. Resources are usually single memory cells (variables) or whole blocks of memory. Deadlocks occur, if all running threads wait for a resource to be released by another

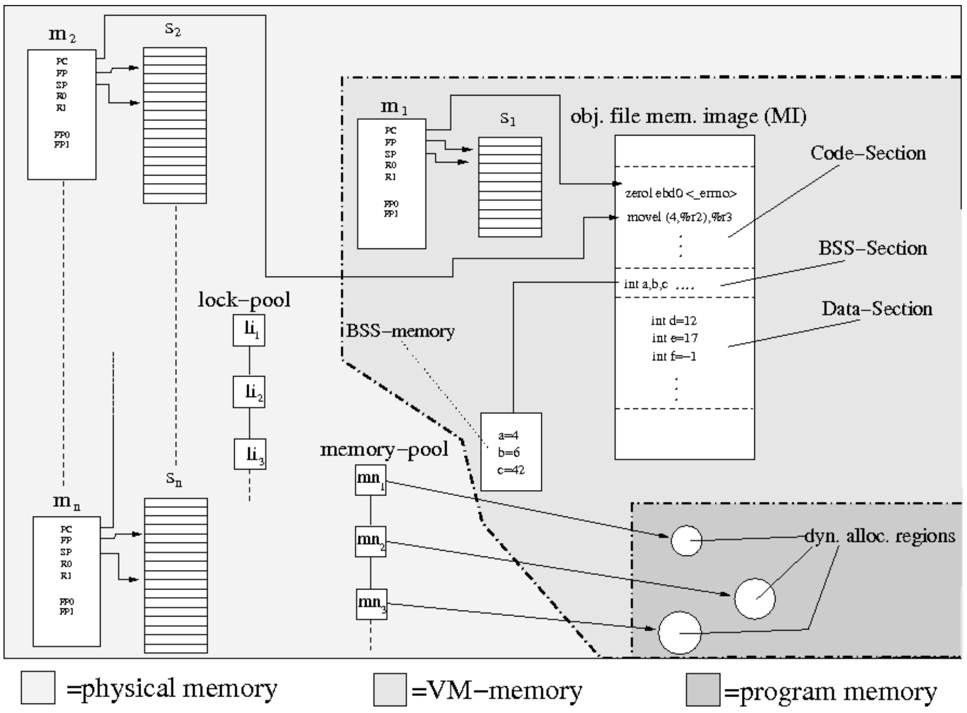

Figure 4.2: System state of a *StEAM* program.

thread, while privilege violations are caused by the attempt to read, to write or to unlock a resource that is currently locked by another thread.

Memory: The state should include information about the location and size of dynamically allocated memory blocks, as well as the allocating thread. Figure 4.2 shows the components that form the state of a concurrent program for *StEAM*.

Enhancing the state of IVM to the state of StEAM involves various changes:

Memory Layers: First of all, the memory is now divided in three layers: The outermost layer is the physical memory which is visible only 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 stacks must refer to the state immediately after the last

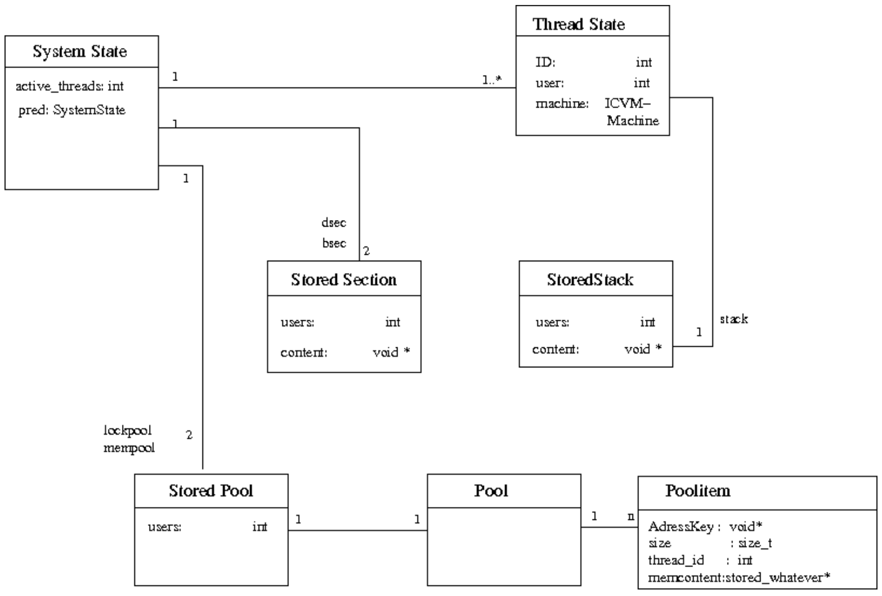

Figure 4.3: Class diagram of StEAM's state description.

execution of the same thread, or if it is new, directly after initialization.

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.

4.3.2 The State Description

Figure 4.3 depicts the diagram of classes which encapsulate the system state in StEAM.

For the memory-efficient storing of states, we rely on the concept of *stored components*. That is, for components that will remain unchanged by many state transitions, we define an additional container class. During successor generation we do not store copies of unchanged components in the system state. Instead, we store a pointer to the container of the respective component in the predecessor state and increment the *user* counter for the component. The user counter is needed when states are deleted. In this case the user counter of each component is decreased and its content is deleted only if the counter reaches zero.

On the top level of our state description, we have the class *System State*, which memorizes the number of running threads and a pointer to its predecessor state in the search tree.

A system state relates to a set of *Thread States* - one for each running thread. Each thread state contains the unique ID of the thread it corresponds to, a user counter and an instance of *ICVMMachine* - the structure which encapsulates the state of CPU-registers in IVM. Note that we do not use a container class for the latter, as the machine registers constitute the only system component that is assured to change for each state transition.
Furthermore a thread state relates to a *Stored Stack* - the container class for storing stack contents. During initialization, IVM allocates 8 MB of RAM for the stack. However, we only need to explicitly store that portion of the stack in our state which lies below the stack pointer of the corresponding thread.
A thread state also relates to two objects of type *Stored Section*. This class defines the container for the contents of the Data- and BSS-section.
Finally, we have two relations to objects of type *Stored Pool* - the container class for the lock- and memory-pool.

4.3.3 Command Patterns

Command patterns form the base technology that enables us to enhance and control IVM. A command pattern is basically a sequence of machine instructions in the compiled code of the program. Rather than being executed, such a pattern is meant to provide the model checker with information about the program or the investigated properties. A command pattern begins with a sequence of otherwise senseless instructions that will not appear in a real program. The C++ code:

```
{
   int i;
   i++;
   --i;
   i++;
   --i;
   i=17;
   i=42;
}
```

Declares a local variable i and in- and decrements it twice. The code compiles to the following sequence of machine instructions:

```
a9de ecfd    inc1l (-532,%fp)
75de ecfd    dec1l (-532,%fp)
a9de ecfd    inc1l (-532,%fp)
75de ecfd    dec1l (-532,%fp)
ebe4 1100    ecfd   movewl #0x11,(-532,%fp)
ebe4 2a00    ecfd   movewl #0x2a,(-532,%fp)
```

The numbers before each instructions' mnemonic denote the opcode and parameters in Little-Endian notation. The number -532 or $fdec$ hexadecimal is an offset in the stack frame used to address the local variable. This offset varies depending on the state of the program. As a first enhancement, IVM was taught to understand command patterns. Concretely, this means that between the execution of two instructions, IVM checks if the value of the next four opcodes indicate a command pattern. In this case, the next four instructions are skipped which is equivalent to increasing the PC by 16 bytes. For the following two instructions, IVM reads the first parameter (i.e. the 2-byte value at offset 2 from the increased PC) but also skips the execution of the actual opcodes. As a result, the values 17 and 42 have been determined as the parameters of the command pattern. The fist parameter indicates the type of information transmitted by the pattern. The second parameter gives the value of this information. For example the type may be information about the source code line which corresponds to the current position in the machine code and the value may give the concrete line number.

In StEAM, command patterns are generated through parameterized C++-macros, which compile to the respective sequence of machine instructions. The macros are then used to annotate the source code of the respective program. Depending on the pattern type, this is either done by the user or automatically by a script, which is executed prior to the model checking. Command pattern serve various purposes, including:

File Information This information tells the model checker the name of the source file which a block of machine code was compiled from.

Line Information The information about the source code line, which the PC of the executed thread's state corresponds to. This is - among other reasons - needed to construct the trail for a detected error.

Capturing the Main Thread The main thread refers to the part of the program that instantiates and starts instances of user-defined thread-classes. The beginning of the *main()* method in the main thread is recognized by IVM through the first line information pattern which occurs in the compiled code.

Locking and Unlocking Exclusive access to a memory region is claimed and released by lock and unlock macros. These compile to command patterns which tell IVM the address and size of the respective region.

Atomic Regions The user can use macros to mark atomic regions. These are executed as a whole without switching to another thread. The user can declare sections of code as atomic regions if they are known to be correct, which can significantly truncate the state space of the program.

Errors IVM can be notified about a program error through a respective command pattern. This pattern is only reached, if a previous logical expression over the program variables evaluates to false (see also section 4.4).

Non-Determinism Model checking requires that the investigated program is not purely deterministic, but that it can take several paths while being executed. For multi-threaded programs, non-determinism arises from the possible orders of thread executions. In StEAM, an additional degree of non-determinism is given by the use of statements, which tell IVM to nondeterministically choose a variable value from a discrete range of the variable's domain. The user can induce non-determinism into the program by annotating the source code using macros. These compile to command patterns, which are recognized by IVM. Formally, we can say, a nondeterministic statement σ, affecting variable x in state s, transforms s into m states $s[x = \sigma(1)], \ldots, s[x = \sigma(m)]$, by replacing the content of variable x with values $\sigma(1), \ldots, \sigma(m)$.

4.3.4 Multi-Threading

As previously mentioned, standard C++ does not support multi-threading. It was therefor decided to implement a custom multi-threading capability into IVM. On the programming level, this is done 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. From the running threads, new threads can be created dynamically. *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).

IVM recognizes new threads through a command pattern in the run-method of a thread class. The pattern is generated by an automated annotation of the program source code.

4.3.5 Exploration

In the following we describe how StEAM explores the state space of a program. We first illustrate the steps needed to generate a successor state. Then we discuss the supported search algorithms and the hashing in StEAM. We close with an example on how a simple concurrent program is model checked by StEAM and an illustration of its search tree.

State Expansion

According to section 4.3.1, a state in StEAM is described by a vector

$$(m_1, \ldots, m_n, s_1, \ldots, s_n, BS, DS, MP, LP),$$

where m_i and s_i denote the machine registers and stack contents of a thread, the components BS, DS are the state of the BSS- and Data-sections, and MP, LP the state of the memory- and lock-pool.

State Expansion is performed in several steps:

1. Initialize the set of successor states $\Gamma \leftarrow \emptyset$.

2. Choose one of the running threads.

3. Restore the contents of the CPU-registers, stack, and the variable sections in the physical addresses.

4. Execute a state transitions - i.e. execute a single instruction or an atomic block of instructions starting at the PC of the thread.

5. Read the contents of the physical adresses to generate a single-element set of successor states $\Gamma' \leftarrow \{t\}$

6. If a nondeterministic statement σ with m possible choices was executed, set $\Gamma' \leftarrow \Gamma' \backslash \{t\} \cup \{t[x = \sigma(1)], ..., t[x = \sigma(m)]\}$.

7. Set $\Gamma \leftarrow \Gamma \cup \Gamma'$.

8. Repeat the steps 2-7 for all running threads.

Algorithms

StEAM provides several state exploration algorithms, which determine the order in which the states of a program are enumerated. Besides the uninformed algorithms depth-first search (DFS) and breadth-first search (BFS), the heuristic search algorithms best-first and A* are supported (cf. Chapter 3).

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 proved to significantly reduce the memory requirements of a model checking run. The method is similar to the *Collapse Mode* used in Spin [Hol97c]. 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 storage of two identical components is therefore possible. Additional savings may be gained through reduction techniques like heap symmetry [Ios01], which are subject to further development of *StEAM*.

Glob

Figure 4.4 shows a simple program `Glob`, which generates two threads from a derived thread class `MyThread`, that access a shared variable `glob`. Note, that including the main program this results in three running threads.

```
#include "IVMThread.h"
#include "MyThread.h"
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;
```

```
#include <assert.h>
#include "MyThread.h"
#define N 2

class MyThread;
MyThread * t[N];
int i,glob=0;

void initThreads () {
 BEGINATOMIC
   for(i=0;i<N;i++) {
     t[i]=new MyThread();
     t[i]->start();
    }
 ENDATOMIC
}

void main() {
 initThreads();
 VASSERT(glob!=8);
}
```

Figure 4.4: The source of the program `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 [Hol97b]. 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 4.5 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.

The assertion is only violated, if the `run` 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, error trails found with

```
Step 1: Thread 1 -     28:   ENDATOMIC
Step 2: Thread 3 -     29:   glob=(glob+1)*ID;
Step 3: Thread 3 -     30: }
Step 4: Thread 2 -     29:   glob=(glob+1)*ID;
Step 5: Thread 2 -     30: }
Step 6: Thread 1 -     29: }
Step 7: Thread 1 -     33:  VASSERT(glob!=8);
```

Figure 4.5: The error-trail for the 'glob'-program.

DFS can become very long, in some cases even too long to be traceable by the user.

The Search Tree

Figure 4.6 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 shapes. Ovals symbolize whole states. DATA is depicted as a diamond, circles denote BSS. 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 4.4, this calls and executes the *initThreads* method in one atomic step. As the result $S_{1,1}$ has three running threads. DATA is modified, because the main thread writes to the un-initialized class pointers $t[i]$. BSS remains unchanged. The stack of the main thread is changed by using the local variable i. The memory pool is changed, because the call of the *new*-constructor during thread generation allocates memory for the instances of the *MyThread*-class. The CPU registers of the main thread change, because they include the program counter, which was altered by the state transition. 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 BSS. All other components - with the exception of the respective thread state - remain unchanged.

4.4 Summary of StEAM Features

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

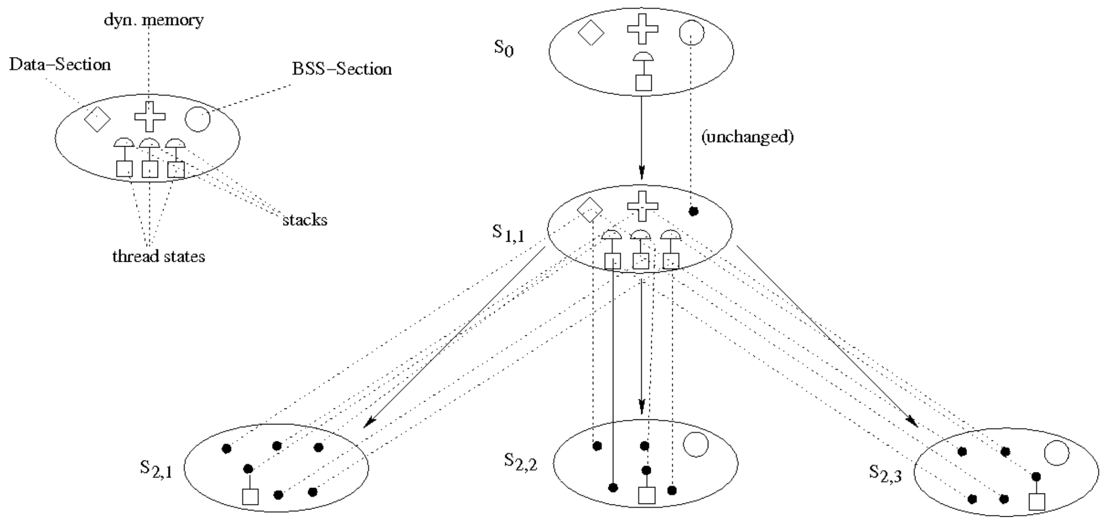

Figure 4.6: The search tree of the glob example.

4.4.1 Expressiveness

Due to the underlying concept which uses a virtual machine to interpret machine code compiled by a real C/C++ compiler[3] 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.

4.4.2 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 4.4. If in the near future a standard for multi threading will arise, *StEAM* may be enhanced to support it.

4.4.3 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[4].

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.

[3]IVM uses a modified version of GCC to compile its code.
[4]A following semicolon is optional.

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 terminates.

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.

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 a privilege violation, and the error trail is returned.

4.4.4 Applicability

StEAM offers new possibilities for model checking software in several phases of the development 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, the 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.

4.4.5 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 [ML03].

4.4.6 Detecting Illegal Memory Accesses

An illegal memory access (IMA) constitutes a read- or write operation to a memory cell that neither lies within the current stack frame, nor within one of the dynamically allocated memory regions, nor in the static variable sections. On modern operating systems, IMAs are usually caught by the memory protection which reports the error (e.g., *segmentation fault* on Unix) and terminates the program.

IMAs are not only one of the most common implementation errors in software, they are also among most time consuming factors in software development as these errors tend to be hard to detect.

On *Wikipedia* [5], the free online encyclopedia, the term *segmentation fault* is exemplified with the following small c program.

```
int main(void)
{
  char *p = NULL;
  *p = 'x';
  return 0;
}
```

When the program is compiled with gcc on Linux, and then being run the operation system reports a segmentation fault. Despite the minimalistic program, the error is not immediately obvious (at least to less experienced programmers), as it results from a wrong view of the level of variable p.

Invoking StEAM to the the (IVM-)compiled code of the program, returns the output:

```
   .
   .

Illegal memory write, printing trail!
depth: 2
Step 1: Thread 1 -     19:    char *p = 0;
Step 2: Thread 1 -     20:    *p = 'x';
```

[5]http://en.wikipedia.org/

```
   .
   .
```

This is already more informative than the lapidary ”segmentation fault” returned by the operating system when the program is merely executed. If we insert a line as follows:

```
 int main(void)
 {
   char *p = NULL;
   p=new int[1];
   *p = 'x';
   return 0;
 }
```

and re-run StEAM on the compiled code, we get

```
   .
   .

(unique) States generated        : 5
Maximum search depth:            : 5
   .
   .
Verified!
   .
   .
```

This implies, that StEAM could enumerate all (five) states of the program without finding an error.

Though nice for illustration purposes, the example is not overly impressive and one may claim, that no model checker was needed for find the error. StEAM however aims at finding less trivial errors that do not occur immediately at the beginning of the program.

A common practice of programmers is to manually annotate the source code with a large amount of *printf* statements and monitor the output of the program to track down the error. This is a tedious practice as it requires the user to run the programs several times while gradually adding more statements in a binary-search fashion until the exact line of the error was found. In contrast, StEAM only needs a single run on the inspected program to provide the user with that information. Moreover, in the first case the *printf*'s become redundant after the error was found and have to be removed in another tedious pass.

More advanced users may rather use a debugger like *gdb* to detected illegal memory accesses. Although preferable to the printf-method, a debugger is still limited in the amount of information it can give to the user. That information merely consists of the stack trace of the error state s - i.e. the memory contents in s and the local variable contents of s and its calling functions up to the top level (usually the *main* function). In contrast, StEAM memorizes the complete path from the initial state of the program to s. This also includes the CPU registers and memory contents of all preceding states, which enables the user to precisely analyse the history of the error.

Moreover, StEAM is able to detect illegal memory accesses that may pass unnoticed by both, the operating system's memory protection and by debuggers. Consider the following program:

```
void main(int argc, char ** argv) {
   int i[100];
   i[500]=42;
}
```

Here, it is attempted to access an element outside of a locally declared array. This error is exclusively reported by StEAM, since it checks, that all write accesses to the stack must reside within the current stack frame, while the system's memory protection merely checks, whether the respective memory cell lies inside an allocated memory region.

Checking for illegal memory access is a time-consuming task, since before the execution of any machine instruction it must be checked, whether it accesses a memory cell and whether this cell resides within the current stack frame, a dynamically allocated memory region or in the static variable sections. Hence, StEAM must be explicitly advised to search for illegal memory accesses through a parameter.

4.5 Heuristics

The use of heuristics constitutes one of if not *the* most effective way to improve the effectiveness of a model checker. Despite the state explosion problem, empirical evaluations show, that the minimal depth of an error state in the search tree often grows only linearly in the number of processes. Thus, the model checker only needs to explore a small fraction of the entire state space to find the error. Heuristics help to explore those paths first, that are more likely to lead to an error state.

Heuristic search algorithms, such as *best-first* search or A* [HNR68] (cf. Chapter 3) can be used in model checking to accelerate the search for errors, to reduce memory consumption, and to generate short and comprehensible counter examples (trails). Note that heuristics are suitable only for the finding of errors, and not for proving correctness, since the latter requires an exhaustive exploration of the program.

Heuristics for model checking generally differ from heuristics for other state space searches, as we usually do not have a complete description of an error state. Exceptions are trail-directed heuristics, like Hamming- or FSM-distance [EM03], which use a two-stage search for shortening suboptimal trails. The lack of information about the searched state makes it hard to devise estimates that are admissible or even consistent .

In the following, we discuss the heuristics currently supported by StEAM. Some of them are derived from the heuristic explicit state model checker HSF-Spin [ELL01] and from the Java Byte Code model checker JPF [GV02], others are new.

4.5.1 Error-Specific Heuristics

Here, the term "error-specific" refers to the class of heuristics, which are targeted to finding a certain type of error.

Most Blocked A typical example for an error-specific heuristic is *most-blocked* [GV02, ELL01, LL03], which was designed to speed up the search for deadlocks. Most-blocked takes the number of non-blocked processes as the estimated distance to the error state. A process is blocked, if it waits for access to a resource.

The most-blocked heuristic is consistent, if a state transition can at most block one process. In StEAM, this is assured, if the investigated program does not contain atomic regions, as a LOCK statement (cf. Section 4.3.1) can merely block the executed thread and only LOCK statements can block threads.

If atomic blocks are present, the most-blocked heuristic may become inadmissible: Consider a state u with processes $p_1, .., p_n$ $(n > 2)$, none of which is blocked, hence $h(u) = n$. For thread t_1, the next two actions are to lock a resource r_1, then a resource r_2. All other processes will lock r_2 first, then r_1. In a transition $u \to v$, thread t_1 locks r_1. In another, transition $v \to w$, thread t_2 locks r_2. Thus, w is a deadlock state, as all threads are mutually waiting for either r_1 or r_2 to be released. This means, the real distance of u to an error state is 2.

Lock and Block For StEAM, an improved version of most-blocked was devised, namely *lock-and-block*. This heuristic uses the number of non-blocked processes plus the number of non-locked resources as the estimated distance to the error. Here, 'locked' means that a process has requested or holds the exclusive access to a resource. Clearly, a deadlock occurs, if all processes wait for a lock to be released by another process. Hence locks are an obvious precondition for threads to get in a blocked state. As the experimental results in Chapter 8 will show, lock-and-block constitutes a significant improvement over the most-blocked heuristic.

Obviously, Lock and Block is inadmissible as deadlocks do not require all available resources to be locked. Nevertheless, the heuristic is more informative than most-blocked and hence preferable over most-blocked - the experimental results in Chapter 8 will support this claim.

Formular-Based Heuristic This heuristic aims to speed up the search for violations of properties given by a logical formula f. The value $h(s)$ estimates the minimal number of transitions needed to make f true (or to make $\neg f$ true, respectively). The heuristic was devised and implemented for the SPIN-based heuristic model checker HSF-Spin [ELL01]. An integration into StEAM or other assembly-level model checkers would be difficult, as the heuristic requires full information about how a logical expression (e.g.

in an assertion) relates to a variable's name and its value, while the compiled machine code operates on stack offsets and memory cells in the DATA- and BSS-sections.

4.5.2 Structural Heuristics

In contrast to error-specific heuristics, structural heuristics [GV02] are not designed to find a certain type of error. Instead, they exploit structural properties of the underlying programming language to speed up the finding of errors in general. Two examples of structural heuristics are the *interleaving* and *branch-coverage* heuristics that were devised to be used in the Java PathFinder.

Structural heuristics differ from other heuristics in so far, as they prefer states that maximize a function m, rather than states that minimize the estimated distance to an error. The function m may e.g. correlate with the degree of thread interleaving. To fit into the concept of heuristic search algorithms such as best-first, which demand an estimated error distance of some state u, we need to subtract $f(u)$ from a constant c, which is hopefully always greater than $f(u)$ for all states u. Obviously, this makes most structural heuristics inadmissible.

Interleaving With interleaving, those paths are explored first, which maximize the number of thread interleavings. Many program errors are caused by the illegal interleavings of threads. For instance, in a block of instructions, a thread may read the content (e.g. the value 1000) of a memory cell c (account balance) into a processor register, perform some arithmetic operation to the register (e.g. add 100) and write the result back to c. If between the reading and the writing, another process is executed, which changes c (subtract 50), the contents of c become wrong (c now contains 1100 although it should be 1050). Thus, maximizing the interleaving of threads, may speed up the finding of such concurrency errors.

Branch-Coverage With the branch-coverage heuristic the paths are favored, that maximize the number of branch instructions which were executed at least once. A greater branch coverage also implies a greater source coverage of the program including those parts that may cause an error.

StEAM supports a subset of JPFs structural heuristic as well as some new ones. Besides lock-and-block, the new heuristics of StEAM regard the access to memory cells and the activeness of threads.

Read-Write The read-write heuristic simply counts the number of reads and writes to any memory cells. States that maximize the number of reads and writes are preferred in the exploration. The intuition behind the heuristic is obvious: Accesses to memory cells imply changes to program variables. As safety and reachability properties (cf. Section 2.1.6) in programs usually relate to variable values, more changes increase the probability of an error.

Alternating Access The alternating access heuristic prefers paths, that maximize the number of subsequent read and write accesses to the same memory cell in the global variable sections. The intuition is, that changes in a global variable x by one thread may influence the behavior of another thread that relies on x. For example, a thread may read the contents of x into a CPU register r, and perform some algebraic operations on r and write the result back to x. If in between another thread changes the content of x, a data inconsistency arises.

Threads Alive The number of threads alive in a program state is a factor that can safely be combined with any other heuristic estimator. Obviously, only threads that are still alive can cause an error.

Chapter 5

Planning vs. Model Checking

Automated planning has established itself as an important field of research in computer science. Indicators for this are events such as the annual *International Conference on Automated Planning and Scheduling* (ICAPS) [1] and the biennial *International Planning Competition* [2].

Research on planning has created sophisticated techniques that aim to alleviate the search for (preferably optimal) plans. This is all the more interesting, as the fields of planning and model checking are highly related and achievements in one field such as heuristics and pruning techniques can easily be carried over from one field to the other. In fact, model checking tasks can straightforwardly be represented as planning problems and vice versa. For example [Ede03] proposes a conversion from the input language of the model checker SPIN to an action planning description language, which makes protocols accessible to action planners.

Consequently, this chapter makes a disgression to the domain of planning and its relation to model checking. We will point out similarities and differences between planning and model checking and discuss how the two fields of research can benefit from each other. Finally, to prove the claimed relation between the two fields, we show an application of the model checker StEAM for solving instances of the $n^2 - 1$-puzzle and multi-agent planning problems.

Planning generally refers to the finding of a plan to reach a goal - i.e., the finding of an operator sequence, which transforms an initial state into a goal state. The simplest class of planning problems is called *propositional planning*. A *propositional planning problem* (in STRIPS notation) is a finite state space problem $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, where $\mathcal{S} \subseteq 2^{AP}$ is the set of states, $\mathcal{I} \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states, and $\mathcal{O}$ is the set of operators that transform states into states. Operators $o = (P, A, D) \in \mathcal{O}$ have propositional preconditions P, and propositional effects (A, D), where $P \subseteq AP$ is the *precondition list*, $A \subseteq AP$ is the *add list* and $D \subseteq AP$ is the *delete list*. Given a state S with $P \subseteq S$, its successor $S' = o(S)$ is defined as $S' = (S \setminus D) \cup A$.

For an operator $o = (P, D, A)$, let $pre(o) = P$, $del(o) = D$ and $add(o) = A$. A *plan* for the propositional planning problem Π is a sequence of operators $o_1, .., o_m$, implying

[1]http://www.icaps-conference.org/

[2]http://ls5-www.cs.uni-dortmund.de/ simedelkamp/ipc-4/

a sequence of states $\mathcal{I} = s_1, .., s_{m+1} \supseteq \mathcal{G}$, such that for $i = 1, .., m$, $pre(o_i) \subseteq s_i$ and $s_{i+1} = (s_i \setminus del(o_i)) \cup add(o_i)$.

Extensions to propositional planning include *temporal planning*, which allows parallel plan execution. In temporal planning, each operator o has a duration $dur(o)$. In a parallel plan execution, several operators $o_1, .., o_l$ may be executed parallel in state s, if all preconditions hold in s and if the effect of one operator does not interfere with the preconditions or effect of another operator - i.e.: For all $i = 1, .., l$: $pre(o_i) \subseteq s$ and for all $i \neq j$: $del(o_i) \cap pre(o_j) = \emptyset \wedge del(o_i) \cap add(o_j) = \emptyset$. A common goal in temporal planning is to minimize the *makespan* of a plan. In parallel plans, each of its operator o has a time $\tau(o)$ at which the operator starts. The makespan of a parallel plan $o_1, .., o_m$ is defined as the point $max_{o \in \{o_1, .., o_n\}} \tau(o) + dur(o)$, when all operators are completed. The case study in section 5.5 refers to planning in a multi-agent environment, where we want to minimize the makespan of a plan, that gets a set of jobs done.

5.1 Similarities and Differences

Search An obvious similarity of planning and model checking is that both methods rely on the exploration of state spaces. In model checking, the state space describes the set of configurations of a system model or program, while in planning the states constitute sets of propositions which are true in a state. Like transitions in model checking lead from system configurations to system configurations, operators in planning lead from sets of propositions to sets of propositions.

Path Both, planning and model checking, return a sequence of state transitions (operators) to the user. In model checking, a path from the initial configuration to an error state serves as a counter example (or error trail), which alleviates the task to find the source of the error in the program or system specification. In planning, the returned sequence of operators describes a plan which leads from the initial state to the designated target state.

Quality In both technologies the returned path can be of different quality. In model checking, the quality of a counterexample is usually negatively correlated with its length, since shorter trails are in general easier to track by the user. The quality of a plan depends on the respective domain and the criterion that is to be optimized. For instance, if the criterion is the costs of the plan and each operator implies the same costs, a plan with less operators is always better. Alternatively, we may want to minimize the makespan of a parallel plan. In this case, a plan with more operators can be better.

Plan/error finding vs verification The goal of planning is always to find a plan - desirably the optimal one. An optimal plan can be obtained by applying an admissible search strategy to the respective problem description. For instance, if we want to minimize the number of operators, a plan returned by breadth-first search is always optimal. When such a plan is found, the remaining state space of the planning problem is of no further interest.

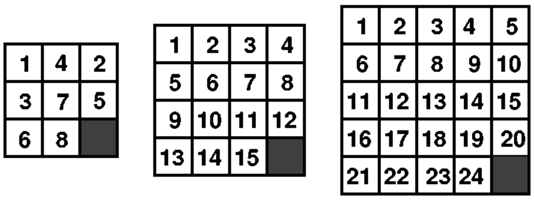

Figure 5.1: The Eight-, Fifteen-, and Twenty-Four-Puzzle.

The same applies, when we search for errors in programs. For instance, in the implementation phase of a program, the presence of errors is very probable. Hence the goal is to find one error at a time and correct it with the help of the returned error trail. Here, directed search with appropriate heuristics can help to reduce the number of states visited until an error is found. At a later point, e.g. before the software is shipped, we may be interested in the *verification* of a program. This means, we need to do an exhaustive enumeration of all possible system states, while each visited state is checked for errors. At this point, the use of directed search is not longer appropriate, as it merely prioritizes the order in which the states are visited. Also, on a per-state base, heuristic search is usually computationally more expensive than undirected search methods, such as depth-first or breadth-first search.

Planning via Model Checking The goal in the $n^2 - 1$-*puzzle* (a.k.a. *Sliding Tile Puzzle*) is to arrange a set of $n^2 - 1$ numbered tiles from left to right and top to bottom in ascending order. As the only operation, the *blank* may switch positions with one of it's neighbouring tiles. Figure 5.1 illustrates examples for $n = 3, 4, 5$.

The paper [HBG05] exploits a PDDL encoding of the $n^2 - 1$ puzzle which can be solved by established planners. Likewise, it is easy to formulate the puzzle as a C++ program, such that instances can be solved by StEAM. The main program is depicted in Figure 5.2. The program accepts as its parameters the scale n followed by a the initial arrangement of the tiles from left to right and top to bottom. The parsing and initialization is done in one atomic step. Afterwards, an infinite loop is executed which first tests, if the tiles are arranged in their goal configuration through an assertion. More precisely, the assertion is violated if and only if the goal state of the puzzle instance was reached. If the assertion holds, i.e., if at least one tile is not at its final position, the program uses a nondeterministic statement to choose one of the functions *ShiftUp*, *ShiftDown*, *ShiftLeft* and *ShiftRight*, which exchange the blank with one of its neighboring tiles. Figure 5.3 lists the function *ShiftUp* - the other three functions are implemented analogously.

We can invoke StEAM with breadth-first search to the instance of the 8-puzzle depicted in Figure 5.1 by:

```
steam -srctrl -BFS npuzzle 3 1 4 2 3 7 5 6 8 0 | grep Shift
```

The output of the model checker is pipelined to the Unix command *grep* such that only the calls to the *Shift* methods in the returned error trail are displayed. The resulting output is:

```
Step 8: Thread 1 -     96:       ShiftLeft();
Step 17: Thread 1 -     90:       ShiftUp();
Step 26: Thread 1 -     90:       ShiftUp();
Step 35: Thread 1 -     96:       ShiftLeft();
```

As you can easily verify, this is the shortest ”plan” to solve the puzzle instance.

As a second example, we present a method, which uses StEAM as a planner in concurrent multi-agent systems. In contrast to classical planning, we avoid the generation of an abstract model. The planner operates on the same compiled code that controls the actual system.

```
int blank;
int n;
short tmp;
short * tiles;

void main(int argc, char **argv) {

  int i=-1;

  BEGINATOMIC;

  if(argc<6) {
    fprintf(stderr, "Illegal number of Arguments!\n");
    exit(1);
  }
  n=atoi(argv[1]);
  if(argc!=n*n+2) {
    fprintf(stderr, "Illegal Number of Arguments\n");
    exit(1);
  }

  tiles=(short *) malloc(n*n*sizeof(short));
  s=(Shifter **) malloc(4*sizeof(Shifter *));
  for(i=0;i<n*n;i++) {
    tiles[i]=atoi(argv[i+2]);
    if(tiles[i]==0) blank=i;
  }

   ENDATOMIC;

  while(1) {
    for(i=0;i<n*n && tiles[i]==i;i++);
    VASSERT(i<n*n);
    RANGE(i,0,3);
    switch(i) {
    case 0:
      ShiftUp();
      break;
    case 1:
      ShiftDown();
      break;
    case 2:
      ShiftLeft();
      break;
    case 3:
      ShiftRight();
      break;
    }
  }
```

Figure 5.2: Main Program of the C++ Encoding of the $n^2 - 1$ puzzle.

```
void ShiftUp() {
  BEGINATOMIC;
  if(blank>=n) {
    tmp=tiles[blank-n];
    tiles[blank-n]=0;
    tiles[blank]=tmp;
    blank-=n;
    }
   ENDATOMIC;
   }
```

Figure 5.3: The method *ShiftUp* of the C++ Encoding of the $n^2 - 1$ puzzle.

5.2 Related Work

Software model checking is a powerful method, whose capabilities are not limited to the detection of errors in a program but to *general problem solving*. For instance, [EM03] successfully uses JPF to solve instances of the sliding-tile puzzle. The adaption is simple: action selection is incorporated as non-deterministic choice points into the system.

Symbolic model checkers have been used for *advanced AI planning*, e.g. the Model-Based Planner (MBP) by Cimatti et al.[3] has been applied for solving *non-deterministic* and *conformant planning* problems, including *partial observable state variables* and *temporally extended goals*.

An architecture based on MBP to *interleave plan generation and plan execution* is presented in [BCT03], where a planner generates conditional plans that branch over observations, and a controller executes actions in the plan and monitors the current state of the domain.
The power of *decision diagrams* to represent sets of planning states more efficiently has also been recognized in *probabilistic planners*, as the SPUDD system [HSAHB99] and its *real-time heuristic programming* variant based on the LAO* algorithm [HZ01] show. These planners solve *factored Markov decision process problems* and encode probabilities and reward functions with *algebraic decision diagrams*.

TL-Plan [BK00] and the TAL planning system [KDH00] apply *control pruning rules*, specified in first order temporal logic. The hand-coded rules are associated together with the planning problem description to accelerate plan finding. When expanding planning states the control rules for the successors are derived by *progression*.

Planning technology has been integrated in existing model checkers, mainly by providing the option to accelerate error detection by heuristic search [ELLL04]. These efforts are referred to as *directed model checking*. First model checking problems have been au-

[3]http://sra.itc.it/tools/mbp

tomatically converted to serve as *benchmarks* for international planning competitions[4].

The work described in [AM02] reduces job-shop problems to finding the shortest path in a *stopwatch automaton* and provides efficient algorithms for this task. It shows that model checking is capable to solve hard combinatorial scheduling problems.

The paper [DBL02] describes a translation from Level 3 PDDL2.1 to *timed automata*. This makes it possible to solve planning problems with the *real-time model checker* UPPAAL. The paper also describes a case study about an implementation of the translation procedure which is applied to the PDDL description of a classical planning problem.

Some work on *multiagent systems* shares similarities with our proposal. In [dWTW01] a framework called ARPF is proposed. It allows agents to exchange resources in an environment which fully integrates *planning and cooperation*. This implies, that there must also be an *inter-agent communication*.

The work [BC01] proposes model checking for *multi-agent systems*, with a framework that is applied to the analysis of a relevant class of multi-agent protocols, namely *security protocols*. In a case study the work considers a belief-based exploration of the *Andrew Authentication Protocol* with the model checker nuSMV that is defined by a set of propositions and evolutions of message and freshness variables.

Multi-agent planning is an AI research of growing interest. A forward search algorithm [Bre03] based on the single-agent planner *FF* that solves multiagent problems synthesizes partially ordered temporal plans and is described in an own formal framework. It also presents a general *distributed algorithm* for solving these problems with several coordinating planners.

The paper [PvV$^+$02] is closely related to the work at hand. It describes *ExPlanTech*, as an enhanced implementation of the *ProPlanT* multiagent system used in an actual industrial environment. *ExPlanTech* introduces the concept of *meta-agents*, which do not directly participate in the production process, but observe how agents interact and how they carry out distributed decision making. The meta-agent in *ExPlanTech* serves a similar purpose as the model checker in our system, as it is able to induce efficiency considerations from observations of the community workflow. Still, the meta-agent uses its own abstract model of the system.

The approach we consider in this paper differs from all the above in that it uses *program model checking* and that it is *applied to multiagent systems* in order to *interleave planning and execution* and *learn* from an existing executable of a given implementation.

5.3 Multi Agent Systems

Modern AI is often viewed as a research area based on the concept of *intelligent agents* [RN95]. The task is to describe and build agents that receive signals from the environment. In fact, each such agent implements percepts to actions. So distributed *multiagent systems* have become an important sub-field of AI, and several classical AI topics are now broadly studied in a concurrent environment. *Planning for multiagent systems* extends classical AI Planning to domains where several agents act together. Application areas

[4]http://ipc.icaps-conference.org

include multirobot environments, cooperating Internet agents, logistics, manufacturing etc. Approaches differ for example in their emphasis on either the *distributed plan generation* or the *distributed plan execution* process, and in the ways communication and perception is used.

The largest discrepancy between generating a plan and executing it, is that multiagent planners usually cannot directly work on existing programs that are executed. Here, we show how StEAM can be utilized to serve as a planning *and* execution unit to improve the performance of concurrent multiagent system implementations.

Directing the exploration towards the planning goal or specification error turns out to be one of the chances to tackle the *state explosion problem*, that arises due to the combinatorial growth of system states. As StEAM works with different object-code distance metrics to measure the efforts needed to encounter the error, we can also address the *evaluation problem* that multiagent systems or multirobot teams have. By having access to object code we can attribute execution time to a set of source code instructions.

The goal is to develop a planning system which interacts with the software units to improve the quality of the results. Since the performance of the system on test increases over time, our approach considers *incremental learning*. As a special feature, the planner should not build an abstract model of the concurrent system. Instead, planning should be feasible directly on the implementation of the software units. This introduces *planning on the source-code and assembly-level*.

5.4 Concurrent Multiagent Systems in StEAM

We regard a *concurrent multiagent system* as a set of homo- or heterogeneous autonomous software units operating in parallel. Some components of the system are shared among all units - such as tasks or resources. The units cooperate to reach a certain goal and the result of this cooperation depends on what decisions are made by each agent at different times.

Concurrent multiagent systems can be implemented into StEAM in a straightforward fashion. Software units are represented as threads. Each unit is implemented by an instance of a corresponding C++-class. Each such class is derived from StEAM's thread-class *IVMThread*. After creating a class instance, a call to the *start*-method adds it to the list of concurrent processes running in the system. The shared components of the system are represented by global variables.

Model checking in StEAM is done by performing exploration on the set of system states, which allows to store and retrieve memorized configurations in the execution of the program. In the simulation mode e.g. controlled by the user, by random choices, or by an existing trace, the model checker executes the program along one sequence of source code instructions.

Here, StEAM serves two purposes. First as a platform to *simulate* the system, second as a *planner* which interacts with the running system to get better results. We assume an *online scenario*, where planning has to be done in parallel to the execution of the software units. In particular, it is not possible to find a complete solution in advance.

We assume that the planner performs a *search on system states*, which are of the same type as that of the actual concurrent system. As a result, the planner finds a desired target state t, which maximizes the expected solution quality. State t will not necessarily fulfill the ultimate goal, because an exhaustive exploration is in general not possible due to time and space restrictions. As the next step, the planner must carry over the search results to the *environment*. However - in contrast to the plan generation - the planner does not have full control over the actual system, due to non-deterministic factors in the environment, such as the execution order of the agents. Instead, the planner must communicate with the software units to influence their behavior, so that the actual system reaches state t or a state that has the same or better properties as t w.r.t. the solution quality.

5.4.1 Interleaving Planning and Simulation

We want to integrate the required planning ability into the model checker with minimal changes to the original code. First, a model checker operates on sequential process executions, rather than on parallel plans. To address this issue, the concept of *parallel steps* is added to StEAM. A parallel step means that all active processes are iterated one atomic step in random order. With parallel steps, we can more faithfully simulate the parallel execution of the software units.

Communication is realized by a concept we call *suggestion*. A suggestion is essentially a component of the system state (usually a shared variable). Each process is allowed to pass suggestions to the planner (model checker) using a special-purpose statement SUGGEST. The SUGGEST-statement takes a memory-pointer as its parameter, which must be the physical address of a 32-bit component of the system state. The model checker collects all suggestions in set SUG. When the model checker has finished the plan generation, it overwrites the values of all suggestions in the current state of the actual environment with the corresponding values in target state t. The software units recognize these changes and adapt their behavior accordingly. Formally, the role of suggestions can be described as follows: Let V be the set of variables that form a state in our system. A system state is defined as a function $s : V \to I\!R$, which maps variables to their domains. Let s be the root state of a planning phase and t the designated target state. Then, the subsequent simulation phase starts at state:

$$s' = s \setminus \left(\bigcup_{v \in Sug} \{v \mapsto s(v)\} \right) \cup \bigcup_{v \in Sug} \{v \mapsto t(v)\}.$$

A concrete example will be given in the case study in Section 5.5.

5.4.2 Combined Algorithm

The proposed system iterates two different phases: *planning* and *simulation*. In the *planning phase* all units are frozen in their current action. Starting from a given system state, the model checker performs a breadth-first exploration of the state space until a time limit is exceeded or no more states can be expanded. For each generated state u an

```
Procedure Interleave
Input: The initial state $s$ of the system, time limit $\theta$, evaluation function $h$
1. loop
2.   $besth \leftarrow h(s); t \leftarrow s; open \leftarrow \{s\}$
3.   while $(time < \theta \wedge open \neq \emptyset)$ /* start of planning phase */
4.        $u \leftarrow getNext(open)$
5.        $\Gamma \leftarrow expand(u)$ /* expand next state in horizon list */
6.        for each $v \in \Gamma$
7.             if $(h(v) > besth)$ $t \leftarrow v$; $besth \leftarrow h(v)$
8.   for each $\sigma \in SUG$
9.        $s.\sigma \leftarrow t.\sigma$ /* write suggestions */
10.  $c \leftarrow 0$
11.  while $(\neg c)$ /* begin of simulation phase */
12.       for each agent $a$: $s \leftarrow iterate(s, a)$ /* do a parallel step */
13.       $c \leftarrow evaluateCriterion()$
```

Figure 5.4: The Implementation of the Interleaved Planning and Simulation System.

evaluation function value $h(u)$ is calculated, which measures the solution quality in u. When a time limit θ is reached, the state with the best h-value is chosen as the target state t and the *simulation phase* starts. At the beginning of the simulation phase, for each element in the set of suggestions, the model checker overwrites the value of the corresponding component in s with the value in t. Then parallel steps are executed in s until the criterion for a new planning phase is met.

Algorithm *Interleave* in Figure 5.4 illustrates the two phases of the system, and is based on top of a general state expanding exploration algorithm. Besides the initial state the exploration starts from, it takes the preference state evaluation function as an additional parameter. A time threshold stops planning in case the model checker does not encounter optimal depth. For the ease of presentation and the online scenario of interleaved execution and planning, we chose an endless loop as a wrapper and have not included a termination criterion in case the goal is encountered.

The function *evaluateCriterion* determines, whether a new planning phase should be initiated or not. The function is not defined here, because this criterion varies depending on the kind of system we investigate. The function *iterate(s,a)* executes one atomic step of an agent (thread, process) a in state s and returns the resulting state.

Figure 5.5 shows how the generated system states switch from a *tree structure* in the planning phase to a *linear sequence* in the simulation phase. We see that the simulation (path on right side of the figure) is accompaigned by intermediate planning phases (tree structure, top left and bottom right of the figure). The search tree contains the (intermediate) target state t - i.e. the generated state with the highest h-value. As we will see, state t is potentially - but not necessarily traversed in the simulation phase.

Ideally, the two phases run in parallel, that is the running multiagent system is not interrupted. Since we use the same executable in our model checker to simulate and plan,

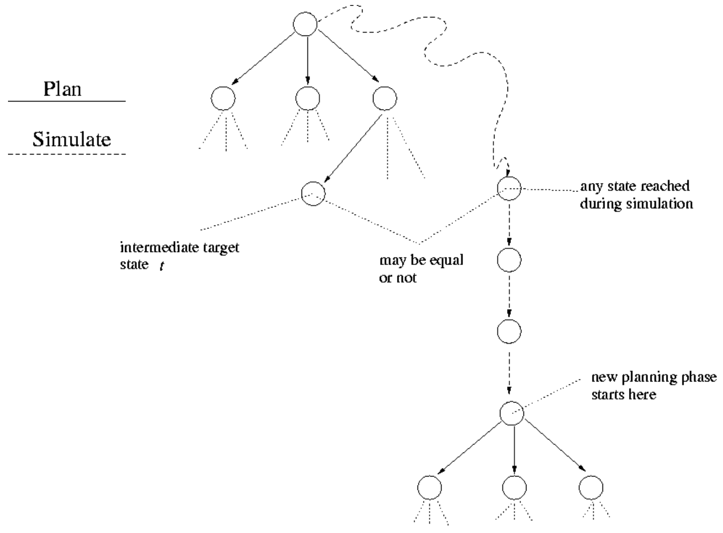
Plan
Simulate
any state reached during simulation
intermediate target state t
may be equal or not
new planning phase starts here

Figure 5.5: State generation in the two different exploration phases.

we store system states that are reached in the simulations and *suspend its execution*. The next planning phase always starts with the last state of the simulation, while the simulation continues with the root node of the planning process. The information that is learned by the planning algorithm to improve the execution is stored in main memory so it can be accessed by the simulation.

5.5 Case Study: Multiagent Manufacturing Problem

In the following we formalize a special kind of a concurrent multiagent system, which is used to evaluate our approach. A *multiagent manufacturing problem*, *MAMP* for short, is a six-tuple (A, J, R, req, cap, dur), where $A = \{a_1, \ldots, a_n\}$ denotes a set of agents, $J = \{j_1, \ldots, j_m, \diamond\}$ is a set of jobs including the empty job $\diamond$, and $R = \{r_1, \ldots, r_l\}$ is a set of resources. The mappings req, cap, and dur are defined as follows:

- $req : J \rightarrow 2^R$ defines the resource requirements of a job,
- $cap : A \rightarrow 2^R$ denotes the capabilities of an agent, and
- $dur : J \rightarrow I\!N$ is the duration of a job in terms of a discrete time measure with $dur(\diamond) = 0$.

A *solution* to a MAMP $m = (A, J, R, req, cap, dur)$, is a function $sol : A \rightarrow 2^{I\!N \times J}$. For $(t, \iota) \in I\!N \times J$, let $job((t, \iota)) = \iota$ and $time((t, \iota)) = t$. Furthermore, let $alljobs_{sol} : A \rightarrow 2^J$ be defined as $alljobs_{sol}(a) = \bigcup_{s \in sol(a)} job(s)$. We require sol to have the following properties.

i. For each $sol(a) = \{(t_0, \iota_0), \ldots, (t_q, \iota_q)\}$ we have that $i \neq j$ implies either $(\iota_i = \iota_j = \diamond)$ or $\iota_i \neq \iota_j$ and for each $i \geq 0$ we have $t_{i+1} \geq t_i + dur(\iota_i)$.

ii. For each $s \in sol(a)$: $req(job(s)) \subseteq cap(a)$.

iii. For all $a \neq a'$ and all $s \in sol(a), s' \in sol(a')$ with $req(job(s)) \cap req(job(s')) \neq \emptyset$ we have either $time(s) + dur(job(s)) \leq time(s')$ or $time(s) \geq time(s') + dur(job(s'))$

iv. For all $a \neq a'$ we have $(alljobs_{sol}(a)\backslash\{\diamond\}) \cap alljobs_{sol}(a') = \emptyset$.

v. Last but not least, we have $\bigcup_{a \in A} alljobs_{sol}(a) = J\backslash\{\diamond\}$.

Property *i.* demands, that each agent can do at most one job at a time. Property *ii.* says, that an agent can only do those jobs, it is qualified for. Property *iii.* means, that each resource can only be used by one agent at a time and properties *iv.* and *v.* demand, that each job is done exactly once.

An *optimal solution sol* minimizes the *makespan* τ until all jobs are done:

$$\tau(sol) = \max_{a \in A, s \in sol(a)} (time(s) + dur(job(s)).$$

MAMPs are extensions to *job-shop scheduling* problems as described e.g. in [AM02]. As the core difference, MAMPs also have a limited set of agents with individual capabilities. While in a job-shop problem one is only concerned with the distribution of resources to jobs, MAMPs also require that for each job we have an agent available that is capable to use the required resources.

5.5.1 Example Instance

Apparently, the solution quality of a MAMP relates to the degree of parallelism in the manufacturing process. A good solution takes care that each agent works around the clock - if possible. Consider a small MAMP $m = \{A, J, R, req, cap, dur\}$ with $A = \{a_1, a_2\}$, $J = \{j_1, j_2, j_3\}$, $R = \{r_1, r_2, r_3, r_4, r_5\}$, $req(j_1) = \{r_1, r_2\}$, $req(j_2) = \{r_3, r_4\}$, and $req(j_3) = \{r_3, r_5\}$. Furthermore $cap(a_1) = R$, $cap(a_2) = \{r_3, r_4, r_5\}$, and $dur(j_1) = 2$, $dur(j_2) = dur(j_3) = 1$.

Figure 5.6 illustrates a best, an average and a worst case solution for m. Here, we have a time line extending from left to right. For each solution, the labeled white boxes indicate the job, which is performed by the respective agent at a given time. Black boxes indicate that the agent is idle. In the optimal solution, a_1 starts doing j_1 at $time = 0$. Parallel to that, a_2 performs j_2 and afterwards j_3. The total time required by the optimal solution is 2. Note, that the solution does not contain any black boxes. In the average solution, which is in the middle, a_1 executes j_2 at $time = 0$ and j_1 afterwards. This implies that a_2 has to be idle up to $time = 1$, because it is incapable to use r_1 needed for j_1 and r_3 is used by j_1. Then, a_2 can at least commence doing j_3 at $time = 1$ when r_3 is available again. The time needed by the average solution is 3. Finally, in the worst solution, which is the bottom-most in Figure 5.6, a_1 first performs j_2, then j_3 and j_1, while a_2 is compelled to be idle at all time. The worst solution needs 4 time units to get all jobs done.

5.5.2 Implementation

For the chosen concurrent multiagent manufacturing system, our goal is to develop a planning procedure for an instance that interacts with the multiagent system to *maximize the level of parallelism* and thus to minimize the idle times of agents, which is

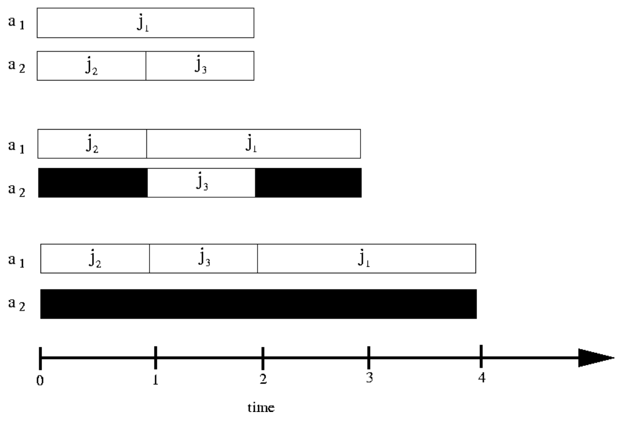

Figure 5.6: Three different solutions for the same MAMP m.

expected to improve the quality of the resulting MAMP-solution. Additionally, as an immediate result of our proposal, we avoid the generation of an abstract model of the system. Instead, we plan directly on the actual implementation.

Information about the state of the jobs and the available resources are stored in global variables. When running, each agent a automatically searches for available jobs he is qualified for and executes them. If a finds a free job j, such that $req(j) \subseteq caps(a)$ and all resources in $req(j)$ are available, the agent requests the resources and starts executing the job. After finishing the job - i.e. when $dur(j)$ time units have passed, a will release the resources and search for the next job. If a cannot find a job, he waits one time unit and searches again. In the case of our multiagent system, each agent passes as a suggestion the local variable *assigned*, which stores the next job to be executed by the agent. Before an agents autonomously seeks a job, it checks if the default-value of its *assigned*-variable has changed. This implies, that - as a result of the previous planning phase - the model checker has assigned a job for this agent. If this is the case, the agent skips the procedure, which looks for a feasible job and starts allocating the required resources.

5.6 Search Enhancements

When using breadth-first search, for n active processes (agents), the number of expanded states in the planning phase is bounded by $O(n^d)$ where d is the depth of the search tree. Without search enhancements, it is hard to reach a search depth which is deep enough to gain information for the simulation phase. As a result, the solutions obtained through the combination of planning and simulation would not be any better, than that of a purely autonomous system without planning. Several pruning and refinement techniques help our planning system to reach a larger search depth.

5.6.1 State Space Pruning

First of all, the model checker StEAM uses a *hash table* to store the already visited states. When a state is expanded, only those successor states are added to the *Open* list, which are not already in the hash table.

Second for a multiagent management system, it does not make sense to generate those states of the search tree that correspond to an execution step of a job. Such an execution only changes local variables and thus does not influence the behavior of the other agents. To realize this pruning, threads can announce themselves *busy* or *idle* to the model checker using special-purpose statements. In the case of the MAMP, an agent declares himself busy, when he starts working on a job and idle when the job is finished. When expanding a state, only those successors are added to the *Open* list that correspond to the execution of an idle thread.

Third, an idle agent looks for a job in one atomic step. Therefore, the execution of an agent that does not result in a job assignment yields no new information compared to the predecessor state. For the given multiagent system, this implies that we only need to store those successor states that result in the *change of a suggestion* (i.e. in a job assignment).

5.6.2 Evaluation Functions

As shown in Figure 5.5 the planner cannot always enforce the target state t to be reached, due to *clashes* in the simulation phases. A clash occurs, if an agent a is iterated before another agent b and autonomously picks a job assigned to b. Our approach considers the value of suggestions to reflect the individual choices of each agent - in this case the job it will do next. The value of this choice is determined by the agent or the planner, before the job itself is taken. Since an agent has no information about the internal values of other agents, clashes are inevitable. This raises the question, what the planner can rely on as a result of its interaction with the system. The answer to this lies in the choice of the evaluation function that is used to determine the intermediate target state during the planning phase. In our case, we use the number of allocated jobs as the evaluation function h, where *allocated* means that an agent is working on that job. It holds, that $h(s) \leq h(s')$ for any successor state of s, since the counter of assigned jobs never decreases - even if a job is finished. In other words h describes the number of currently allocated, plus the number of finished jobs.

Theorem 5.1. *Let t be the intermediate target state in the search tree. Furthermore, let s_i describe the i-th state of the simulation phase and p_i denote the i-th state on the path from the initial state to t in the search tree, where $i \in \{0, \dots, m\}$ and $p_m = t$. Then for each $0 \leq i \leq m$ we have that $h(s_i) \geq h(p_i)$.*

Proof. In fact, we have $h(s_0) = h(p_0)$ and $h(s_1) \geq h(t)$. The equality $h(s_0) = h(p_0)$ is trivial, since the two states are equal - except for the values of the suggestions. This implies that in particular, the variable which counts the number of assigned jobs has the same value in s_0 and p_0. Furthermore all suggestions in s_0 are initialized with the values from t. If we now consider the first simulation step, there are two possibilities:

In the first case, we have no clashes during the parallel step performed between s_0 and s_1. This implies that each agent picks the job assigned by the model checker (if any). So we have at least $h(s_1) = h(t)$. Additionally, agents that have no job in t may pick an unassigned job, so that $h(s_1) > h(t)$.
In the second case we assume to have clashes. For each such clash, we have one agent, that cannot pick its assigned job which decreases $h(s_1)$ by one. However, we also have another agent, that picks a job, although it has no job in t, which increases $h(s_1)$ by one. So, a clash does not influence the value of $h(s_1)$ and thus we have $h(s_1) \geq h(t)$.

Altogether, we have $h(s_0) = h(p_0)$, $h(s_1) \geq h(t)$ and since the number of taken jobs never decreases, it holds that for all $1 \leq i \leq m$, we have $h(s_i) \geq h(s_{i-1})$ and $h(p_i) \geq h(p_{i-1})$ and in particular $h(t) \geq h(p_{i-1})$, which implies $h(s_i) \geq h(p_i)$ for all $0 \leq i \leq m$. $\square$

Experimental results for multi agent planning can be found in section 8.5.

A main insight of the case study is that assembly model checking allows us to develop a planning methodology, which does not require the construction of an abstract model of the environment. Instead, planning is performed directly on the compiled code, which is the same as used by the simulation. Also, it shows that the capabilities of StEAM are not limited to verification of software.

In the future, we would like to try the approach on actual multiagent systems - be it pure software environments or physical systems like a group of robots. Certainly, many multiagent systems are more complex than the MAMP architecture presented here as they e.g. also involve inter-agent communication. We kept our framework simple for deriving a multiagent prototype to test our planning approach.

In the long term, both areas may highly benefit from the assembly model checking approach, because it eliminates the task of building an abstract model of the actual system or program. Not only does this save a considerable amount of resources, usually spent on constructing the models, but performing model checking and planning on the actual implementation also avoids possible inconsistencies between the model and the real system.

Chapter 6

Hashing

Hashing is essential in software verification, since concurrent programs exhibit a large number of duplicate states. In fact, it turns out that ignoring duplicates in StEAM bars the model checker from finding errors even in very simple programs due to memory restrictions.

Hashing is not heavily discussed in other fields related to state space search - simply because the portion of hashing compared to the overall computational effort of the search is negligible. In this context, program model checking takes a special position among the universe of state space problems, since hashing shows to be the computationally most expensive operation when expanding a state. This can be explained by the exceptionally large state description of a program (cf. Section 4.3.1). Most state transitions of a program relate to a single statement or a short sequence of statements that only change a small fraction of the state, e.g., the memory cell of a variable. Since only the changed parts need to be stored, generating a successor state is computationally inexpensive if we consider the size of the underlying state description. In contrast, conventional hash functions process the entire state, and as a consequence computing the hash code for each newly generated state dramatically slows down the exploration.

This makes the design of efficient hash functions a central issue of program model checking. An obvious solution to overcome the complexity stated above, is to devise an *incremental* hash function, i.e. one that computes the hash code of a state relative to that of it's immediate predecessor by looking at the changes caused by the corresponding transition. Incremental hashing is used in other areas, such as text search. However, existing approaches rely on linear structures with a fixed size. In contrast, a program state is inherently *dynamic*, as new memory regions may be allocated or freed. Also, it is *structured*, in the sense that the state description is subdivided in several parts, such as the stacks and the global variables.

The following chapter is dedicated to the design of an incremental hash function on dynamic, structured state descriptions that can be implemented into StEAM. We first discuss hashing in general. We then proceed by devising and incremental hash function for static linear state vectors. The approach is exemplified for AI puzzles and action planning. Finally, we enhance our approach to be usable for dynamic and structured state descriptions. The contribution of incremental hashing to the field of program model checking is experimentally evaluated using an implementation in the StEAM model checker in chapter 8.

6.1 Hash Functions

A hash function $h : \mathcal{S} \to \mathcal{K}$ maps states from a universe $\mathcal{S}$ to a finite set of keys $\mathcal{K}$ (or codes). A good hash function is one that minimizes the number of *collisions*. A hash collisions occurs, if for two states $s \neq s' \in \mathcal{S}$, $h(s)$ is equal to $h(s')$. Since, the number of possible state configurations is usually greater than $|\mathcal{K}|$ (sometimes infinite), hash collisions cannot be avoided. In state space search, hashing is used to compute a table address on which to store a generated state or to lookup a previously stored one. As a necessary criterion, to minimize the number of hash collisions, the function must be parameterized with all state components.

Distribution To achieve a uniform distribution of the key mapped to the set of addresses, a random experiment can be performed. Computers, however, can only simulate random experiments, aiming at number sequences whose statistical properties deviate as less as possible from the uniform distribution. The *Lehmer-generator* is a pseudo random number generator on the basis of linear congruence, and is one of the most common methods for generating random numbers. A sequence of pseudo-random numbers x_i is generated according to $x_0 = b$ and $x_{i+1} = (ax_i + c)$ *mod* m for $i \geq 0$. A good choice is the *minimal standard generator* with $a = 7^5 = 16,807$ and $m = 2^{31}-1$. To avoid encountering overflows, one uses a factorization of m.

Remainder Method If one can extend $\mathcal{S}$ to $\mathbb{Z}$, then $\mathcal{T} = \mathbb{Z}/m\mathbb{Z}$ is the *quotient space* with equivalence classes $[0], \ldots, [m-1]$ induced by the relation

$$z \sim w \textit{ iff } z \textit{ mod } m = w \textit{ mod } m.$$

Therefore, a mapping $h : \mathcal{S} \to \{0, 1, \ldots, m-1\}$ with $h(x) = x$ *mod* m distributes $\mathcal{S}$ on $\mathcal{T}$. For the uniformity, the choice of m is important: for example, if m is even then $h(x)$ is even if and only if x is. The choice $m = r^w$, for some $w \in \mathbb{N}$, is also not appropriate, since for $x = \sum_{i=0}^{l} a_i r^i$ we have

$$x \textit{ mod } m = \left(\sum_{i=w}^{l} a_i r^i + \sum_{i=0}^{w-1} a_i r^i \right) \textit{ mod } m \quad = \quad \left(\sum_{i=0}^{w-1} a_i r^i \right) \textit{mod } m$$

This means that the distribution only takes the last w digits into account.

A good choice for m is a prime which does not divide a number $r^i \pm j$ for small j, because $m \mid r^i \pm j$ is equivalent to $r^i \textit{mod } m = \mp j$ so that (case $+$)

$$x \textit{ mod } m = j \cdot \sum_{i=0}^{l} a_i \textit{ mod } m,$$

i.e., keys with same (alternating) sum of digits are mapped to the same address.

6.2 Explicit Hashing

Explicit hashing schemes store the entire state description, without loss of information. In practice, this is usually realized using a pointer table H of size m. For state s with $h(s) = c$, it is first tried to store s at $H[c]$. If $H[c]$, already holds a state s', it is determined, whether s' is identical to s. In the latter case, s is not stored. Otherwise, we have a hash collision, that has to be resolved. Resolving can be done, by either *probing* or *successor chaining*.

Probing For probing, a function π is used to traverse a sequence of successor positions $H[\pi(c)], .., H[\pi(\pi(c))], ..$ in the hash table. This is done, until a position $H[c']$ is encountered, such that one of the following holds:

- If $H[c']$ is unoccupied, it is used to store s.
- If $H[c']$ holds a duplicate of s, the state is not stored.
- If $c = c'$, the hash table is full. In this case, it must be decided to either not store s, replace another stored state by s or resize the hash table.

Successor Chaining For successor chaining, the hash table entries store a linked list of different states with the same hash value. As an advantage over probing, only states with the same hash value must be compared.

Figure 6.1 illustrates an example of four states s_1, s_2, s_3, s_4 to be inserted into a hash table of size 6 using either probing or successor chaining. Let $h(s_1) = h(s_3) = 2$ and $h(s_2) = h(s_4) = 3$. Furthermore, *linear probing* is used, i.e. $\pi(c) = c + 1$. Obviously, for probing the hashing scheme needs four state comparisons: $s_3 \leftrightarrow s_1$, $s_3 \leftrightarrow s_2$, $s_4 \leftrightarrow s_2$ and $s_4 \leftrightarrow s_3$. In contrast, successor chaining only needs two state comparisons: $s_3 \leftrightarrow s_1$ and $s_4 \leftrightarrow s_2$. Alternatively, to prevent unnecessary comparisons for probing, the hash code can be integrated into the state description. However, this implies an additional memory overhead. As another advantage of successor chaining, the linked list of a hash table entry can grow to an arbitrary size. Therefore, the table never becomes full.

A clear disadvantage of successor chaining is the need for a successor pointer that is stored with each hashed state. If that pointer is integrated into the state description, it raises the memory requirements per state by the size of a memory pointer (4 bytes on a 32-bit system). This implies wasted memory in settings, where states are not explicitly hashed (cf. 6.2.1), as the pointers are never used. Alternatively, one can define a container structure like *hash node*, which holds a pointer to the stored state and to the successor hash node. However, this raises the memory requirements for a stored state by another 4.

Dynamic Arrays As an alternative, which is also used in StEAM, a hash table entry may contain a pointer to an array of states, whose size may dynamically grow, when needed. This constitutes a compromise between the memory-efficiency of probing and the speed of successor chaining.

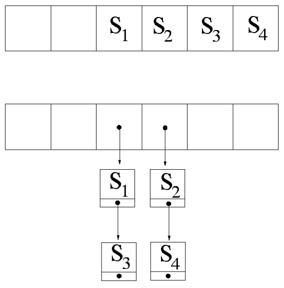

Figure 6.1: Example for explicit hashing with probing (top) and successor chaining (bottom).

Here, when an array in the hash table gets full, its size is doubled. Furthermore - without loss of generality - we assume that a memory pointer takes 4 bytes. Let n be the number of stored states.

Theorem 6.1. *Using dynamic arrays, insertion of one element in the hash table is possible in amortized constant time, while the memory requirements are less or equal than for successor chaining.*

Proof. Assume, we want to insert n elements in the hash table. Let $\{A_1,..,A_r\}$ denote the set of arrays, which hold at least one element. Obviously, we have $r \leq n$.

We start with the proof for the memory requirements. For successor chaining, we always need $4n$ bytes to store the pointer to the element and an additional $4n$ bytes for the successor pointer, which results in a total memory requirement of $8n$ in all cases.

Moreover, let n_i be the number of elements stored in A_i. By definition, the maximal capacity of A_i is $2^{\lceil \log(n_i) \rceil}$ bytes, which implies a total memory requirement of $\sum_{i=1}^{r} 4 \cdot 2^{\lceil \log(n_i) \rceil}$ for dynamic arrays.

In the worst case, all arrays are half full, i.e. $\lceil \log(n_i) \rceil = \log(n_i)$, which yields a memory requirement of $\sum_{i=1}^{r} 8n_i = 8n$ bytes.

In the best case, we have $r = 1$ and there is only one unused element in A_1, i.e. we have $\log(n+1) = \lceil \log(n) \rceil$, which implies a memory requirement of $4n + 4$ bytes. The general observation is, that the memory requirement lies in the interval $[4m + 4, 8m]$.

Next, we prove the amortized constant time. This can be done by an induction over the number of used arrays r.

r=1: The overall time for inserting n elements in a dynamic array consists of the time needed to store the elements and the time needed to resize the array, when it is full. To resize an array from a capacity m to $2m$, the new memory needs to be allocated, the elements must be copied from the old memory and the old memory must be freed. Let c, k, l be constants, where c is the time needed to store an element in the array (without resizing). The constant k shall denote the time required to read an entry from a memory cell and write it to another. Moreover, l constitutes the time needed to allocate memory for the new capacity of an array and to free the old memory. Thus, the time needed to insert n elements is

$$\begin{aligned}
& n \cdot c + \sum_{i=1}^{\log(n)} (2^i \cdot k + l) \\
= & \sum_{i=1}^{\log(n)} 2^i \cdot c + \sum_{i=1}^{\log(n)} (2^i \cdot k + l) \\
= & n \cdot c + n \cdot (k + l) \\
= & n \cdot (c + k + l)
\end{aligned}$$

This means, the insertion of one element in the hash table is possible in amortized constant time $c + k + l$.

r→r+1: For $r+1$ we have one more array in which $n' \leq n$ elements are inserted. For each of the n' elements that fall in the new array, we know by the induction hypothesis that insertion is possible in amortized constant time t. Moreover it is a trivial observation, that the remaining $n - n'$ insertions into r arrays cause less or equally many array resizings like the insertion of n elements. For the latter, element insertion is possible in amortized constant time t' according to the induction hypothesis. Thus, for $r + 1$ arrays, the insertion of n elements is possible in $n' \cdot t + (n - n') \cdot t'$, which implies that each insertion takes amortized time $t \cdot n'/n \cdot +t' - t' \cdot n'/n \leq t + t'$.

□

6.2.1 Bit-State Hashing

Bit-state hashing [Hol98] belongs to the class of *compacting* or *approximate* hash functions. Instead of storing generated states explicitly, the state is mapped to one or more positions within a bit vector v: Starting with all bits in v set to 0, for each generated state s a set of p independent hash functions is used to compute bit positions $h_1(s), .., h_p(s)$ in v. If all bits at the corresponding positions in v are set to 1, s is considered a duplicate. Otherwise, s inserted into the open list and the corresponding bits in v are set to zero. The approach allows a compact memorization of the set of closed states and is hence particularly efficient in combination with variations of depth-first search where the set of open states is generally small. In consequence, larger portions of the state space can be explored. As a drawback the used algorithm becomes incomplete, as hash collisions lead to false prunings in the search tree which may hide the error state.

Bit-state hashing is also the approach of the widely used model-checker *SPIN* [Hol03], which applies it in the so-called *Supertrace algorithm*.

Let n be the number of reachable states and m be the maximal number of bits available. As a coarse approximation for single bit-state hashing with $n < m$, the average probability P_1 of a hash collision during the course of the search is bounded by $P_1 \leq \frac{1}{n} \sum_{i=0}^{n-1} \frac{i}{m} \leq n/2m$, since the i-th element collides with one of the $i - 1$ already inserted elements with a probability of at most $(i - 1)/m$, $1 \leq i \leq n$. For multi-bit hashing using h (independent) hash-functions with the assumption $hn < m$, the average probability of collision P_h is reduced to $P_h \leq \frac{1}{n} \sum_{i=0}^{n-1} (h \cdot \frac{i}{m})^h$, since i elements occupy at most hi/m addresses, $0 \leq i \leq n-1$. In the special case of double bit-state hashing, this simplifies to

$$P_2 \leq \frac{1}{n}\left(\frac{2}{m}\right)^2 \sum_{i=0}^{n-1} i^2 = 2(n-1)(2n-1)/3m^2 \leq 4n^2/3m^2.$$

6.2.2 Sequential Hashing

An attempt to remedy the incompleteness of partial search is to re-invoke the algorithm several times with different hash functions to improve the coverage of the search tree. This technique, called *sequential hashing*, successively examines various beams in the search tree (up to a certain threshold depth). In considerably large protocol verification problems, Supertrace with sequential hashing succeeds in finding bugs but still returns long error trails. As a rough estimate on the error probability we take the following. If in sequential hashing exploration with the first hash function covers c/n of the search space, the probability that a state x is not generated in d independent runs is $(1-c/n)^d$, such that x is reached with probability $1-(1-c/n)^d$.

6.2.3 Hash Compaction

Like bit-state hashing, the *hash compaction* method [SD96] aims at reducing the memory requirements for the state table. However, it stores a compressed state descriptor in a conventional hash table instead of setting bits corresponding to hash values of the state descriptor. The compression function c maps a state to a b-bit number in $\{0,\dots,2^b-1\}$. Since this function is not surjective, i.e., different states can have the same compression, false positive errors can arise. Note, however, that if the probe sequence and the compression are calculated independently from the state, the same compressed state can occur at different locations in the table.

In the analysis, we assume that breadth-first search with ordered hashing using open addressing is applied. Let the goal state s_d be located at depth d, and $s_0, s_1, \dots, s_d$ be a shortest path to it.

It can be shown that the probability p_k of a false positive error, given that the table already contains k elements, is approximately

$$p_k = 1 - \frac{2}{2^b}(H_{m+1} - H_{m-k}) + \frac{2m + k(m-k)}{m2^b(m-k+1)}, \tag{6.1}$$

where $H_n = \sum_{i=1}^n \frac{1}{i} = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O(\frac{1}{n^4})$ denotes a harmonic number.

Let k_i be the number of states stored in the hash table after the algorithm has completely explored the nodes in level i. Then there were at most $k_i - 1$ states in the hash table when we tried to insert s_i. Hence, the probability P_{miss} that *no* state on the solution path was omitted is bounded by

$$P_{miss} \geq \prod_{i=0}^{d} p_{k_i - 1}.$$

If the algorithm is run up to a maximum depth d, it can record the k_i values online and report this lower bound on the omission probability after termination.

To obtain an *a priori estimate*, knowledge of the depth of the search space and the distribution of the k_i is required. For a coarse approximation, we assume that the table fills up completely ($m = n$) and that half the states in the solution path experience an empty table during insertion, while the other half experiences the table with only one empty slot. This models (crudely) the typically bell-shaped state distribution over the levels $0, \ldots, d$. Assuming further that the individual values in Equation 6.1 are close enough to one to approximate the product by a sum, we obtain the approximation

$$P_{miss} = \frac{1}{2^b}(\ln n - 1.2).$$

Assuming, more conservatively, only one empty slot for all states on the solution path would increase this estimate by a factor of two.

6.3 Partial Search

During the study of approximate hash function, such as *bit-state hashing*, *double bit-state hashing*, and *hash compact*, we have seen that the sizes of the hash tables can be increased considerably. This is paid by a small but affordable lack of search accuracy, so that some synonyms of states can no longer be disambiguated. As we have seen partial search is a compromise to the space requirements that full state storage algorithms have and can be casted as a non-admissible simplification to traditional heuristic search algorithms. In the extreme case, partial search algorithms are not even complete, since they can miss an existing goal state due to wrong pruning. The probability can be reduced either by enlarging the bits in the remaining fingerprint vector or by re-invoking the algorithm with a different hash function.

6.4 Incremental Hashing

If the size of the state description becomes large, a *full* hash function, i.e. one that takes into account every component of the state can considerably slow down the exploration. As a compromise, the hash function may only consider a subset of all state components and risk an increased number of hash collisions. This may be acceptable, if the full state is stored in the hash table. In the worst case, there can be a large set of generated states with the same hash value and the comparison with newly generated states can make the exploration even slower than using a full hash function.

Due to the large state spaces in program model checking, we want bitstate-hashing (cf 6.2.1) to be an option for the exploration. Its use requires that the underlying function produces a minimum of hash collisions, since each collision can result in an unexplored subtree that may contain the error state. To minimize the number of collisions, the hash function must consider all components of the state description.

Incremental hash functions provide both: *fast computation* over large state descriptions and a *good distribution* over the set of possible hash values [EM04]. However, existing applications of incremental hashing, such as Rabin-Karp Hashing (cf 6.5.1) merely rely on statically-sized structures to be hashed. In program model checking however, the size of the state description in inherently dynamic. Also, to best of the authors knowledge, there has been no previous application of incremental hashing to state space search.

We therefore devise an incremental hashing scheme for StEAM in two steps. First, we provide a static framework to apply incremental hashing for arbitrary state exploration problems that involve a state vector of static size. We exemplify these considerations on some classical AI puzzle problems.

After this, we extend the framework to the incremental hashing of dynamic and structured state vectors as they appear in StEAM. An implementation of the dynamic framework is evaluated in chapter 8.

6.5 Static Framework

6.5.1 Rabin and Karp Hashing

The hash computation proposed here is based on an extended version of the algorithm of Rabin and Karp.

For this case, the states in S are interpreted as bit strings and divided into blocks of bits. For example blocks of byte-size yield 256 different *characters*.

The idea originates in matching a text $T[1..n]$ to a pattern $M[1..m]$. In the *algorithm of Rabin and Karp* [KR87], a pattern M is mapped to a number $h(M)$, which fits into a single memory cell and can be processed in constant time. For $1 \leq j \leq n-m+1$ we will check if $h(M) = h(T[j..j+m-1])$. Due to possible collisions, this is not a sufficient but a necessary criterion for the match of M and $T[j..j+m-1]$. A character-by-character comparison is performed only if $h(M) \neq h(T[j..j+m-1])$. To compute $h(T[j+1..j+m])$ incrementally in constant time, one takes value $h(T[j..j+m-1])$ into account, according to Horner's rule for evaluating polynomials. This works as follows. Let q be a sufficiently large prime and $q > m$. We assume that numbers of size $q \cdot |\Sigma|$ fit into a memory cell, so that all operations can be performed with single precision arithmetic. To ease notation, we identify characters in Σ with their order. The algorithm of Rabin and Karp as presented in Figure 6.2 performs the matching process.

The algorithm is correct by the following observation.

Theorem 6.2. *At the start of the j-th iteration we have*

$$t_j = \left(\sum_{i=j}^{m+j-1} T[i] |\Sigma|^{m-i+j-1} \right) \bmod q.$$

Procedure Rabin-Karp
Input: String T, pattern M
Output: Occurrence of M in T

```
    $p \leftarrow t \leftarrow 0$; $u \leftarrow |\Sigma|^{m-1}$ *mod* $q$
    **for each** $i \in \{1, \ldots, m\}$ $p \leftarrow (|\Sigma| \cdot p + M[i])$ *mod* $q$
    **for each** $i \in \{1, \ldots, m\}$ $t \leftarrow (|\Sigma| \cdot p + T[i])$ *mod* $q$
    **for each** $j \in \{1, \ldots, n-m+1\}$
        **if** $(p = t)$
            **if** (*check* $(M, T[j..j+m-1])$) **return** $j$
        **if** $(j \le n-m)$ $t \leftarrow ((t - T[j] \cdot u) \cdot |\Sigma| + T[j+m])$ *mod* $q$
```

Figure 6.2: Algorithm of Rabin and Karp.

```
2  3  5  9  0  2  3  1  4  1  5  2  6  7  3  9  9  2  1
                  \___       ___/      \___       ___/
                      \    /   mod 13     \    /
                       \ /                  \ /
      8  9  3 11  0  1  7  8  4  5 10 11  7  9 11
```

Figure 6.3: Example of hashing for string matching.

Proof. Certainly, $t_1 = \left(\sum_{i=1}^{m} T[i]|\Sigma|^{m-i}\right)$ *mod* q and inductively we have

$$\begin{aligned} t_j &= ((t_{j-1} - T[j-1] \cdot u) \cdot |\Sigma| + T[j+m-1]) \ mod\ q \\ &= \left(\left(\left(\sum_{i=j-1}^{m+j-2} T[i]|\Sigma|^{m-i+j-2}\right) - T[j-1] \cdot u\right) \cdot |\Sigma| + T[j+m-1]\right) mod\ q \\ &= \left(\sum_{i=j}^{m+j-1} T[i]|\Sigma|^{m-i+j-1}\right) mod\ q. \end{aligned}$$

□

As an example take $\Sigma = \{$`0,...,9`$\}$ and $q = 13$. Furthermore, let $M =$ `31415` and $T =$ `2359023141526739921`. The application of the mapping h is illustrated in Figure 6.3. We see that h produces collisions. The incremental computation in the first case works as follows.

$$\begin{aligned} 14{,}152 &\equiv (31{,}415 - 3 \cdot 10{,}000) \cdot 10 + 2\ (mod\ 13) \\ &\equiv (7 - 3 \cdot 3) \cdot 10 + 2\ (mod\ 13) \\ &\equiv 8\ (mod\ 13) \end{aligned}$$

The computation of all hash addresses has a resulting running time of $O(n+m)$, which is also the best case overall running time. In the worst case, the matching is still of order $\Omega(nm)$, as the example problem of searching $M \in 0^m$ in $T \in 0^n$ shows.

6.5.2 Recursive Hashing

Our approach relates to *recursive hashing* [Coh97]. Recursive hash functions consist of a recursive function H and an address function A so that $h(S) = A(H(S))$. The idea of linear recursion is that the contribution of one symbol to the hash value is independent of the contribution of the others. Let T be a mapping from Σ to R, where R is a ring, $r \in R$. Similar to the algorithm of Rabin and Karp we compute the value H of string $S_i = (s_i, \ldots, s_{i+k-1})$ as $H(S_0) = \sum_{j=0}^{k-1} T(s_j)$ and

$$H(S_i) = rH(S_{i-1}) + T(s_{i+k-1}) - r^n T(s_{i-1})$$

for $1 \leq i \leq n$. With this we have $H(S_j) = \sum_{i=0}^{k-1} r^{k-i+j} T(s_{i+j})$.

The *prime-division method* takes $R = \mathbb{Z}$ and assigns $h(S_i) = H(S_i)\ mod\ q$. Here q is chosen as a suitable prime to obtain a good distribution. For faster computation the *two-power division method* can be an option. The computation ignores higher order bits, such that the hash address can be computed by word-level bit-operations.

As a compromise between the bad distribution of *two-power division* and the bad run time behavior of the *prime-division* method is *polynomial hashing*. One can choose ring $R = GF(2)[x]/(x^w - 1)$, where the $GF(2)$ is the Galois-field of characteristic 2. The finite field $GF(2)$ consists of elements 0 and 1 which satisfy the following addition $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 0$, and multiplication $0 \times 0 = 0$, $0 \times 1 = 0$, $1 \times 0 = 0$, and $1 \times 1 = 1$. $GF(2)[x]$ is the polynomial ring with coefficients in $GF(2)$ over $[x]$. A polynomial is represented as a tuple of coefficients. A multiplication with x is a cyclic bit shift, since $x^w - 1$ is equal to the zero-polynomial so that for $q(x) = \sum_{i=0}^{w-1} q_i x^i$ in R we compute

$$xq(x) = \sum_{i=0}^{w-1} q_i x^{i+1} \equiv q_{w-1} + \sum_{i=0}^{w-2} q_i x^{i+1}$$

Instead of $GF(2)[x]/(x^w+1)$ one may choose ring $GF(2)[x]/p(x)$ with $p(x) = x^w + \sum_{i=0}^{w-1} p_i x^i$ being an arbitrary polynomial. In $GF(2)[x]/p(x)$ we have $p(x) \equiv 0$ and for $q = \sum_{i=0}^{w-1} q_i x^i$ we have

$$xq(x) = \begin{cases} \sum_{i=1}^{w-1} q_{i-1} x^i, & \text{if } q_{w-1} = 0 \\ p_0 + \sum_{i=1}^{w-1} (q_{i-1} + p_i) x^i, & \text{if } q_{w-1} = 1 \end{cases}$$

Although some model checkers like SPIN support recursive and polynomial hashing, they still process the entire state vector and are not truly incremental.

6.5.3 Static Incremental Hashing

We regard state vectors with component domains in $\Sigma = \{0, \ldots, l-1\}$. In other words, states are strings over the alphabet Σ. This notational simplification is not a restriction, since the approach generalizes to vectors $v = (v_1, \dots, v_k)$ with $v_i \in D_i$, $|D_i| < \infty$, $i \in \{1, \dots, k\}$. Moreover, for practical software model checking we might expect the state vector to be present in form of a byte array.

A static vector is a vector with fixed length. It may for example represent the data area where the values of global program variables are stored. In the simplest case, only the value of one component is changed. This will be very common in software model checking, since a single statement like $var =< expression >$ will only change the content of the memory cell associated to to *var*. Let $i \in \{1, \dots, k\}$ be the position of the changed component. For a state vector $v = (v_1, \dots, v_n)$ and its successor state $v' = (v'_1, \dots, v'_n)$ with $v'_i = v_i, i \neq j$, we calculate

$$\begin{aligned} h(v') &= \sum_{j=1}^{k} v_j \cdot |\Sigma|^j - v_i \cdot |\Sigma|^i + v'_i \cdot |\Sigma|^i \, mod\, q \\ &= h(v) - v_i \cdot |\Sigma|^i + v'_i \cdot |\Sigma|^i \, mod\, q. \end{aligned}$$

This implies, that $h(v')$ can be computed in constant time. We can generalize the above to the case, where more than one component is changed by the state transition. This occurs, for instance, if several instructions are declared to form an atomic block. These blocks are often used in concurrent programs to prevent a thread switch in critical sections of the code. Let $I(v, v') = \{i \mid v_i \neq v'_i\}$ be the set of indices of all modified components, then

$$\begin{aligned} h(v') &= \sum_{i=1}^{n} v_i \cdot |\Sigma|^i - \sum_{i \in I(v,v')} v_i \cdot |\Sigma|^i + \sum_{i \in I(v,v')} v'_i \cdot |\Sigma|^i \, mod\, q \\ &= h(v) - \sum_{i \in I(v,v')} v_i \cdot |\Sigma|^i + \sum_{i \in I(v,v')} v'_i \cdot |\Sigma|^i \, mod\, q. \end{aligned}$$

Choosing a prime for q is known to provide a good distribution over the set of possible hash codes [Knu98]. This property is also exploited by e.g. the Lehmer-generator [Leh49] to generate pseudo random numbers.

Theorem 6.3. *Computing the hash value of $h(v')$ given $h(v)$ is available in time*

- $O(|I(v, v')|)$*; using $O(k)$ extra space, where $I(v, v')$ is the set of indices that change*
- $O(1)$*; using $O((k \cdot |\Sigma|)^{I_{\max}})$ extra space, where $I_{\max} = \max_{(v,v')} |I(v, v')|$.*

Proof. In the first case, we store $|\Sigma|^i$ *mod* q for all $1 \leq i \leq k$. In the second case, we precompute

$$\sum_{j \in I(v,v')} -v_j |\Sigma|^j + v'_j |\Sigma|^j \text{ mod } q$$

for all possible transitions (v, v'), of which there are at most $\binom{k}{I_{\max}} \cdot |\Sigma|^{I_{\max}} = O((k \cdot |\Sigma|)^{I_{\max}})$ different ones. □

A good hash function should minimize the number of address collisions. Given a hash table of size m and the sequence $k_1, \dots, k_n$ of keys to be inserted, we can define $X_{ij} =$

$1,$ if $h(k_i) = h(k_j)$, and 0, otherwise. Then $X = \sum_{i<j} X_{ij}$ is the sum of collisions. Assuming a random hash function with uniform distribution, we have

$$E(X) = E\left(\sum_{i<j} X_{ij}\right) = \sum_{i<j} E(X_{ij}) = \sum_{i<j} \frac{1}{m} = \binom{n}{2} \cdot \frac{1}{m}.$$

We empirically tested the distribution of incremental hashing on a vector of size 100 and compared the results to randomly generated numbers with $m = 10000019$ (the smallest prime greater than 10000000) and $n = 1000000$. For randomly generated numbers, we got 48,407 collisions, while incremental hashing gave 50,304, an insignificant increase.

6.5.4 Abstraction

Incremental hashing is of particular interest in state space searches, which use abstractions to derive a heuristic estimate, because several hash functions are involved - one for the original state space and one for each abstraction.

State space abstractions ϕ map state vectors $v = (v_1, \dots, v_k)$ to abstract state vectors $\phi(v) = (\phi(v_1), \dots, \phi(v_k))$. This includes

- *data abstraction* [CGP99], which exploits the fact that specifications for software models usually consider fairly simple relationships among the data values in the system. In such cases, one can map the domain of the actual data values into a smaller domain of abstract data values. This induces a mapping of the system states, which in turn induces an abstract system.
- *predicate abstraction* [GS97] , where the concrete states of a system are mapped to abstract states according to their evaluation under a finite set of predicates. Automatic predicate abstraction approaches have been designed and implemented for finite and infinite state systems.

Both data and predicate abstraction induce abstract systems that simulate the original one. Every behavior in the original system is also present in the abstract one.

Pattern databases [CS98, ELL04, QN04] are hash tables for fully explored abstract state spaces, storing with each abstract state the shortest path distance in the abstract space to the abstract goal. They are constructed in a complete traversal of the inverse abstract search space graph. Each distance value stored in the hash table is a lower bound on the solution cost in original space and serves as a heuristic estimate. Different abstraction databases can be combined either by adding or maximizing the individual entries for a state. As abstraction databases are themselves hash tables, incremental hashing can also be applied.

Theorem 6.4. *Let ϕ_i be a mapping $v = (v_1, \dots, v_k)$ to $\phi_i(v) = (\phi_i(v_1), \dots, \phi_i(v_k))$, $1 \le i \le l$. Combined incremental state and abstraction state vector hashing of state vector v' with respect to its predecessor v is available in time*

- $$O\left(|I(\phi(v, v'))| + \sum_{i=1}^{l} |I(\phi_i(v), \phi_i(v'))|\right)$$

using $O(kl)$ extra space, where $I(v,v')$ is the set of indices that change in original space, and set $I(\phi_i(v),\phi_i(v'))$ denotes the set of affected indices in database i

- *$O(l)$; using $O(l\cdot(k\cdot|\Sigma|)^{I_{\max}})$ extra space.*

Proof. Let $h(v)=\sum_{i=1}^{k} v_i|\Sigma|^i$ *mod* q be the hash function for vector v in original space and $h_i(\phi_i(v))=\sum_{j=1}^{k}\phi_i(v_j)\phi_i(|\Sigma|^j)$ *mod* q be the i-th hash function for addressing the abstract state $\phi_i(v)$, $1\le i\le l$, with ϕ_i mapping $v=(v_1,\ldots,v_k)$ to $\phi_i(v)=(\phi_i(v_1),\ldots,\phi_i(v_k))$. In the first case, we store $\phi_i(|\Sigma|^j)$ *mod* q, for $1\le i\le l$ and $1\le j\le k$. In the second case, for all possible transitions (v,v') we precompute

$$\sum_{j\in I(v,v')} -v_j|\Sigma|^j+v'_j|\Sigma|^j \ \mathit{mod}\ q$$

and

$$\sum_{j\in I(\phi_i(v,v'))} -\phi_i(v_j)\phi_i(|\Sigma|^j)+\phi_i(v'_j)\phi_i(|\Sigma|^j),\ \mathit{for}\ i\in\{1,\ldots,l\}.$$

□

6.6 Examples for Static Incremental Hashing

For illustration purposes, we exemplify incremental hashing on static state vectors, before proceeding to considerations about dynamic and distributed state vectors.

6.6.1 Incremental Hashing in the (n^2-1)-Puzzle

Our first example is the (n^2-1)-*Puzzle* presented in Chapter 5. We extend a solver for n^2-1 puzzles that uses IDA* [Kor85] with the Manhattan distance estimate. Let $\Sigma=\{0,\ldots,15\}$. The natural vector representation for state u is $(t_0,\ldots,t_{15})\in\Sigma^{16}$, where $t_i=l$ means that the tile labeled with l is located at position i, and $l=0$ is the blank. The hash value of u is $h(u)=(\sum_{i=0}^{15}t_i\cdot 16^i)$ *mod* q. Let state u' with representation $(t'_0,\ldots,t'_{15})$ be a successor of u. We know that there is only one transposition in the vectors t and t'. Let j be the position of the blank in S and k be the position of the blank in v. We have $t'_j=t_k$, $t'_k=0$, and for all $1\le i\le 16$, with $i\ne j, i\ne k$ it holds that $t'_i=t_i$. Therefore,

$$\begin{aligned}
h(u') &= \left(\left(\sum_{i=0}^{15} t_i\cdot 16^i\right)-t_j\cdot 16^j+t'_j\cdot 16^j-t_k\cdot 16^k+t'_k\cdot 16^k\right)\mathit{mod}\,q\\
&= \left(\left(\left(\sum_{i=0}^{15} t_i\cdot 16^i\right)\mathit{mod}\,q\right)-0\cdot 16^j+t'_j\cdot 16^j-t_k\cdot 16^k+0\cdot 16^k\right)\mathit{mod}\,q\\
&= \Big(h(u)+(t'_j\cdot 16^j)\,\mathit{mod}\,q-(t_k\cdot 16^k)\,\mathit{mod}\,q\Big)\ \mathit{mod}\,q.
\end{aligned}$$

To save time, we precompute $(k\cdot 16^l)\,\mathit{mod}\,q$ for each k and l in $\{0,\ldots,15\}$. If we store $(k\cdot 16^j)\,\mathit{mod}\,q-(k\cdot 16^l)\,\mathit{mod}\,q)$ for each value of j, k, and l, we can save one more addition.

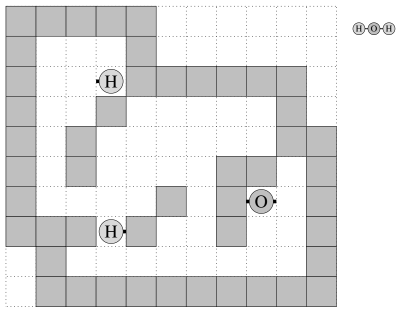

Figure 6.4: One level of Atomix. The depicted problem in can be solved with 13 moves, where the atoms are numbered left-to-right in the molecule: 1 *down left*, 3 *left down right up right down left down right*, 2 *down*, 1 *right*.

As $h(u) \in [0..q-1]$ and $((k \cdot 16^j) \, mod \, q - (k \cdot 16^k) \, mod \, q) \in [0..q-1]$ we may further substitute the last $mod \, q$ operation, by an addition or a subtraction operation.

The savings are larger, when the state description vector grows. For the (n^2-1)-Puzzle non-incremental hashing results in $\Omega(n^2)$ time, while in incremental hashing the efforts remain constant.

The (n^2-1)-*Puzzle* has underwent different designs of heuristic functions. One lower bound estimate is the *Manhattan distance (MD)*. For every two states S and S', it is defined as sum of the vertical and horizontal distances for each tile. MD is consistent, since the differences in heuristic values between two states S and S' are 1, such that $|h(S') - h(S)| = 1$ and $h(S) - h(S') + 1 \geq 0$. The heuristic can be computed incrementally in $O(1)$ time, given a two-dimensional table, which is addressed by the label, the current direction and the position of the tile that is being moved.

6.6.2 Incremental Hashing in Atomix

The goal of*Atomix* [HEFN01] (cf. Figure 6.4) is to assemble a given molecule from atoms. The player can select an atom at a time and *push* it towards one of the four directions left, right, up, and down; it will keep on moving until it hits an obstacle or another atom. The problem is solved when the atoms form the same constellation (the "molecule") as depicted beside the board. Atomix has been proven to be PSPACE complete [HS01]. Note, that (n^2-1)-Puzzle solving is NP complete, while *Sokoban* and STRIPS planning (cf 6.6.3) are PSPACE complete. A concrete Atomix problem, given by the original atom positions and the goal molecule, is called a *level* of Atomix.

For solving Atomix problems we use IDA* search, which compared to A* considerably reduces the memory requirements, while preserving optimality if the underlying heuristic is admissible. The number of duplicates in the search is larger than in regular domains like the (n^2-1)-Puzzle, so state storage will be essential. The design of appropriate state abstractions for abstraction database storage are not trivial. The problem is that by *stopper atoms*, sub-patterns of atoms can be unsolvable while the original problem is not.

For IDA* exploration Atomix has a global state representation that contains both the board layout in form of a two dimensional table and the coordinates for the atoms in form of a simple vector. This eases many computations during successor generation. After a recursive call the changes due to a move operation are withdrawn in a backtrack fashion. Consequently – disregarding the computation time for computing the heuristic estimate, the hashing efforts (computation of the function, state comparisons and copyings), and finding the correct location for an atom in a given move direction – successor generation in Atomix is a constant time operation. We will see that the three remaining aspects can all be implemented incrementally.

Incremental Heuristic A heuristic for *Atomix* can be devised by examining a model with relaxed restrictions. We drop the condition that an atom slides as far as possible: it may stop at any closer position. These moves are called *generalized moves*. Note that the variant of Atomix which uses generalized moves has an undirected search graph. Atomix with generalized moves on an $n \times n$ board is also NP-hard. In order to obtain an easily computable heuristic, we also allow that an atom to slide through other atoms or share a place with another atom. The goal distance in this model can be summed up for all atoms to yield an *admissible* heuristic for the original problem: The heuristic is *consistent*, since the h-values of child states can differ from that of the parent state by 0, $+1$ or -1.

Since each atom attributes one number to an overall sum it is easy to see that the heuristic estimate can be computed incrementally in constant time by subtracting the value for the currently moving atom from its start location and by adding the value for its final destination. We only need to precompute a distance table for each atom.

Incremental Successor Generation For move execution, the implementation, we started with, looked at the set of adjacent squares into the direction of a move unless an obstacle is hit. By the distance of a move between the start and the target location, this operation is not of constant time. Therefore, we looked for alternatives. The idea to maintain a doubly linked *neighbor graph* in x and y direction fails on the first attempt. The neighbor graph would include atoms as well as surrounding walls. When moving an atom, say vertically, at the source location, the update of the link structure turns out to be simple, but when we place it at its target location, we have to determine the position in the linked list for the orthogonal direction. In both cases we are left with the problem to determine the maximal j in a list of numbers that is smaller than a given i. With an arbitrary large set of numbers, the problem is not trivial.

However, we can take advantage that the number of atoms and walls in a given row or column are bounded by a small value (15 in our case). This yields the option to encode all possible layouts in a row or column as a bit-string with integer values in $\{0, \ldots, 2^{15} - 1\}$. For each of the query positions $i \in \{0, \ldots, 15\}$ we store a table M_i of size 2^{15}, with entry $M_i[b]$ denoting the highest bit $j < i$ and $j \geq 0$ in b that is smaller than i. Each table M_i consumes 2^{15} byte or 32 KByte. These tables can be used to determine in constant time the stopping position of an atom that is pushed to the right or downwards starting in row or column i. Analogously, we can build a second set of tables for left and upward pushes. Together, the tables require $2^{15} \cdot 15 \cdot 2$ bytes ($\approx$1MB).

Incremental Hashing A move in Atomix will merely change the position of one atom on the board. Hence a state for a given Atomix level is sufficiently described by the positions $(p_1, \ldots, p_m)$ of atoms on the board. We exploit the fact, that only one atom is changed to calculate the hash value of a state incrementally. This constitutes a special case of the concept described in section 6.4. Let $s = (p_1, \ldots, p_m)$ be a state for an Atomix level. We define its hash value as $h(s) = \left(\sum_{i=1}^{m} p_i \cdot 15^{2i}\right) \mathit{mod}\, q$. The given Atomix solver - *Atomixer* [HEFN01] - uses a hash table of fixed size ts (40MB by default). We adjust this value to the smallest prime that is greater or equal ts and use it as our q to get a better distribution for our hash function [Knu98]. The full calculation of the hash code is only needed for the initial state. Let s' be an immediate successor state which differs from its predecessor s only in the position of atom i. Then we have

$$h(s') = ((h(s) - ((p_i \cdot 15^{2i})\, \mathit{mod}\, q)\, \mathit{mod}\, q) - ((p'_i \cdot 15^{2i})\, \mathit{mod}\, q))\, \mathit{mod}\, q$$

We can use a pre-calculated table t with $15^2 \cdot m$ entries which, for each $i \in \{1, \ldots, 15^2\}$ and each $j \in \{1, \ldots, m\}$, stores $t_{i,j} = (i \cdot 15^{2j})\, \mathit{mod}\, q$. We can use t to substitute $p_i \cdot 15^{2i}$ with $t_{p_i,i}$ and $p'_i \cdot 15^{2i}$ with $t_{p'_i,i}$. Furthermore, we know that $0 \leq h(s) < q$ and apparently $0 \leq a\, \mathit{mod}\, q < q$ for any a. This implies, that each remaining expression taken $\mathit{mod}\, q$ holds a value between $-q+1$ and $2q-2$. As a consequence, each sub-term $a\, \mathit{mod}\, q$ can be substituted by $a - q$ if $a \geq q$, $q + a$ if $a < 0$ or a if $0 \leq a < q$. Now we can calculate $h(s')$ incrementally by the following sequence of program statements:

1. if($t_{p_i,i} > h(s)$) $h(s') \leftarrow q + h(s) - t_{p_i,i}$
2. else $h(s') \leftarrow h(s) - t_{p_i,i}$
3. $h(s') \leftarrow h(s') + t_{p'_i,i}$
4. if($h(s') \geq q$) $h(s') \leftarrow h(s') - q$

This way, we avoid the computationally expensive modulo operators.

6.6.3 Incremental Hashing in Propositional Planning

It is not difficult to devise an incremental hashing scheme for STRIPS planning (cf. 5) that bases on the idea of the algorithm of *Rabin and Karp*. For $S \subseteq AP$ we may start with $h(S) = (\sum_{p_i \in S} 2^i) \bmod q$. The hash value of $S' = (S \setminus D) \cup A$ is

$$\begin{aligned} h(S') &= \left(\sum_{p_i \in (S \setminus D) \cup A} 2^i\right) \bmod q \\ &= \left(\sum_{p_i \in S} 2^i - \sum_{p_i \in D} 2^i + \sum_{p_i \in A} 2^i\right) \bmod q \\ &= \left(\left(\sum_{p_i \in S} 2^i\right) \bmod q - \left(\sum_{p_i \in D} 2^i\right) \bmod q + \left(\sum_{p_i \in A} 2^i\right) \bmod q\right) \bmod q \\ &= \left(h(S) - \sum_{p_i \in D} 2^i + \sum_{p_i \in A} 2^i\right) \bmod q. \end{aligned}$$

Since 2^i mod q can be pre-computed for all $p_i \in AP$, we have a running time that is of order $O(|A| + |D|)$, which is constant for most STRIPS planning problems. It is also possible to achieve constant time complexity if we store the values $inc(o) = (\sum_{p_i \in A} 2^i) \bmod q - (\sum_{p_i \in D} 2^i) \bmod q$ together with each operator. Either complexity is small, when compared to ordinary hashing of the planning state.

Pattern Database Search

Abstraction functions ϕ map states $S = (S_1, \ldots, S_k)$ to patterns $\phi(S) = (\phi(S_1), \ldots, \phi(S_k))$. Pattern databases [CS98] are hash tables for fully explored abstract state spaces, storing with each abstract state the shortest path distance in the abstract space to the abstract goal. They are constructed in a complete traversal of the inverse abstract search space graph. Each distance value stored in the hash table is a lower bound on the solution cost in original space and serves as a heuristic estimate. Different pattern databases can be combined either by adding or maximizing the individual entries for a state.

Pattern databases work, if the abstraction function is a homomorphism, so that each path in the original state space has a corresponding one in the abstract state space. In difference to the search in original space, the entire abstract space has to be looked at. As pattern databases are themselves hash tables we apply incremental hashing, too.

If we restrict the exploration in STRIPS planning to some certain subset of propositions $R \subseteq AP$, we generate a planning state space homomorphism ϕ and an abstract planning state space [Ede01] with states $S_A \subseteq R$. Abstractions of operators $o = (P, A, D)$ are defined as $\phi(o) = (P \cap R, A \cap R, D \cap R)$. Multiple pattern databases are composed based on a partition $AP = R_1 \cup \ldots \cup R_l$ and induce abstractions $\phi_1, \ldots, \phi_l$ as well as lookup hash tables PDB$_1$,…,PDB$_l$. Two pattern databases are additive, if the sum of the retrieved values is admissible. One sufficient criterion is the following. For every pair of non-trivial operators o_1 and o_2 in the abstract spaces according to ϕ_1 and ϕ_2, we have that preimage $\phi_1^{-1}(o_1)$ differs from $\phi_2^{-1}(o_2)$. For pattern database addressing we use a multivariate variable encoding, namely, SAS$^+$ [Hel04].

6.7 Hashing Dynamic State Vectors

In the previous section, we devised an incremental hashing scheme for static state vectors. This is not directly applicable for program model checkers, as they operate on *dynamic* and *structured* states. Dynamic means, that the size of a vector may change. For example, a program can dynamically allocate new memory regions. Structured means, that the state is separated in several subvectors rather than a single big vector. In StEAM for example, the stacks, machines, variable sections and the lock/memory pools constitute subvectors which together form a global state vector. In the following, we extend the incremental hashing scheme from the last section to be applicable for dynamic and distributed states.

For dynamic vectors, components may be inserted at arbitrary positions.

We will regard dynamic vectors as the equivalent of strings over an alphabet Σ. In the following, for two vectors a and b, let a, b denote the concatenation of a and b. For example, for $a = (0, 8)$ and $b = (15)$, we define $a, b = (0, 8, 15)$.

We define four general lemmas for the hash function h as used in Rabin-Karp hashing (cf. Section 6.5.1). Lemmas 1 and 2 relate to the insertion-, lemmas 3 and 4 to the deletion of components. Afterwards, we apply the lemmas to different types of data structures, such as stacks and queues. We use $|a|$ to denote the size of a vector a.

Lemma 1. *For all $a, b, c \in \Sigma^*$ we have $h(a, b, c) = h(a, c) - h(c) \cdot |\Sigma|^{|a|} + h(b) \cdot |\Sigma|^{|a|} + h(c) \cdot |\Sigma|^{|a|+|b|} \, mod \, q$.*

Proof:

$$h(a,c) \;=\; \sum_{i=1}^{|a|} a_i \cdot |\Sigma|^i + \sum_{i=1}^{|c|} c_i \cdot |\Sigma|^{i+|a|} \, mod \, q = h(a) + h(c) \cdot |\Sigma|^{|a|} \, mod \, q.$$

Analogously, we infer the following result.

Lemma 2. *For all $a, b, c \in \Sigma^*$ we have $h(a, b, c) = (h(a, c) - h(a)) \cdot |\Sigma|^{|b|} + h(a) + h(b) \cdot |\Sigma|^{|a|} \, mod \, q$.*

Next, we need to address the removal of components from the vector. Lemma 3 and 4 show a way to incrementally compute the hash value of the resulting vector.

Lemma 3. *For all $a, b, c \in \Sigma^*$ we have $h(a, c) = h(a, b, c) - h(b, c) \cdot |\Sigma|^{|a|} + h(c)/|\Sigma|^{|b|} \, mod \, q$.*

Lemma 4. *For all $a, b, c \in \Sigma^*$ we have $h(a, c) = (h(a, b, c) - h(a, b))/|\Sigma|^{|b|} + h(a) \, mod \, q$.*

Lemmas 3 and 4 require the *multiplicative inverse* of $|\Sigma|^{|b|}$ modulo q. As exploration algorithms often vary the size of their hash table and hence the value of q, the computation of the multiplicative inverse must be performed on run time in an efficient way. This can be achieved using an extended version of the *Euclidean Algorithm* [Sch98] for calculating the greatest common divisor (gcd) of two natural numbers a and b for $a > b$. It computes (d, x, y), where d is the greatest common divisor of a and b, and $d = ax + by$.

It is a known fact [Sch98], that $(\mathbb{Z}_q^*, \cdot_{mod\, q})$ forms a group *if and only if* $\mathbb{Z}_q^* = \{a \in \{1, \ldots, q-1\} \mid gcd(a, q) = 1\}$. If q is a prime, the latter holds for $\mathbb{Z}_q^* = \{1, \ldots, q-1\}$. Hence, by calculating *ExtendedEuclid*(a, q), we get a triple (d, x, y), with $d = gcd(a, q) = 1 = ax + qy$. This implies $ax \, mod \, q = 1$, which means that x is the multiplicative inverse of a.

6.7.1 Stacks and Queues

If the dynamic vector represents a stack- or queue-structure, components are added or removed only at the beginning and the end. For all possible cases, the resulting hash address can be computed incrementally in constant time:

$$\begin{aligned}
h(v_1, \ldots, v_n, v_{n+1}) &= h(v) + v_{n+1} \cdot |\Sigma|^{n+1} \, mod \, q \\
h(v_1, \ldots, v_{n-1}) &= h(v) - v_n \cdot |\Sigma|^n \, mod \, q \\
h(v_0, \ldots, v_n) &= v_0 \cdot |\Sigma| + (h(v) \cdot |\Sigma|) \, mod \, q \\
h(v_2, \ldots, v_n) &= \frac{h(v) - v_1 \cdot |\Sigma|}{|\Sigma|} \, mod \, q = \frac{h(v)}{|\Sigma|} - v_1 \, mod \, q
\end{aligned}$$

The first and second equation refer to insertions and deletions at the end of the vector. The third and fourth equation, address the insertions and deletions at the beginning.

6.7.2 Component Insertion and Removal

If components can be inserted or deleted at an arbitrary position of the vector, a constant-time computation of the hash code is no longer possible. Let the i-th prefix p_i and the i-th suffix s_i of a hash code on a vector $v = (v_1, \ldots, v_n)$ be defined as $p_i(v) = \sum_{j=1}^{i} v_j |\Sigma|^j \, mod\, q$ and $s_i(v) = \sum_{j=i}^{n} v_j |\Sigma|^j \, mod\, q$.
For insertion of a (single) component, we have two alternatives by either computing the prefix or suffix up to the respective position in the vector (Lemma applied in brackets):

$$\begin{aligned} h(v_1, .., v_i, w, v_{i+1}, \ldots, v_n) &\overset{(1)}{=} h(v) - s_i(v) + w \cdot |\Sigma|^{i+1} + s_i(v) \cdot |\Sigma| \, mod\, q \\ h(v_1, .., v_i, w, v_{i+1}, \ldots, v_n) &\overset{(2)}{=} (h(v) - p_i(v)) \cdot |\Sigma| + w \cdot |\Sigma|^{i+1} + p_i(v) \, mod\, q \end{aligned}$$

Analogously, for the removal of a component we have two alternatives:

$$\begin{aligned} h(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) &\overset{(3)}{=} h(v) - s_{i+1}(v) - v_i \cdot |\Sigma|^i + \frac{s_{i+1}(v)}{|\Sigma|} \, mod\, q \\ h(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) &\overset{(4)}{=} \frac{h(v) - p_{i-1}(v) - v_i \cdot |\Sigma|^i}{|\Sigma|} + p_{i-1}(v) \, mod\, q \end{aligned}$$

Clearly, the proposed method will not change the asymptotic runtime of $O(n)$ compared to the non-incremental hashing. However, we need to access only $\min\{i, n-i+1\}$ components, which means $\lfloor n/2 \rfloor$ components in the worst case.

6.7.3 Combined Operations

So far we have only discussed the cases, where either an arbitrary number of components change their value or a single element is being inserted or removed. For completeness, we want to cover also those cases where an arbitrary number of value changes as well as insertions/deletions are applied to the state vector. One solution is to apply the given incremental hash computations consecutively for each set of value changes and each insertion/removal of a component. However, we know that for the removal of one component in a dynamic vector, we must already access $\lfloor n/2 \rfloor$ components in the worst case. If we have to perform several such computations, the time needed for the incremental hash computation will quickly exceed that of a non-incremental hash function. A compromise is to unify all changes to the vector into a single sequence of components such that $v' = av'_1, \ldots, v'_m b$, where a, b are either empty sequences or $a = v_1, \ldots, v_i$ is a prefix and $b = v_k, ..v_n$ is a suffix of v. Then,

$$\begin{aligned} h(v') &= h(v) - s_{i+1}(v) + s_k(v) \cdot |\Sigma|^m + \sum_{j=1}^{m} v'_j \cdot |\Sigma|^{j+i} \, mod\, q \\ &= (h(v) - p_i(v)) \cdot |\Sigma|^m + p_i(v) + \sum_{j=1}^{m} v'_j \cdot |\Sigma|^{i+j} \, mod\, q. \end{aligned}$$

This means, for the general case we need to access $\min\{i+m, n-i+m\}$ elements, which in the worst case equals the length of v' i.e. m. We can expect an improvement if m is small and i or $n-i$ is small - i.e. if v' differs from v only in a small sequence close to one side of the vector.

6.8 Hashing Structured State Vectors

In the preceding sections we have thoroughly addressed the application of incremental hashing in domains, whose state can be described by a single vector of constant size. Based on the previous results, we now devise a incremental hashing scheme for dynamic structured state vectors like those in StEAM.

A structured state vector u can be seen as the concatenation $v_1, \ldots, v_m$ of subvectors $v_i = (v_{i,1}, \ldots, v_{i,n_i})$ for $i \in \{1, \ldots, m\}$. Such a partitioning of v can either be chosen freely or - more naturally - it originates from the underlying state space that is explored. For example, the state of a computer program consists of static components, such as the global variables, as well as dynamic components, like the program stack and the pool of dynamically allocated memory.

6.8.1 The Linear Method

Given an order on the subvectors, we can hash each subvector individually and integrate the results to return the same global hash code as if we hashed the global vector directly. A possible hash function on $v = v_1, \ldots, v_m$ is

$$h(v) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} v_{i,j} \cdot |\Sigma|^{\Sigma_{k=1}^{i-1} n_k + j} \, mod\, q.$$

The runtime needed by a naive incremental algorithm to combine the hash codes of the subvectors into a global hash code varies depending on the changes that were made by the state transition.

A program step - be it a single line of code or some atomic block - usually changes values within a small subset of all allocated memory regions. Hence, we can expect the average runtime to be close to the best case (one affected component) rather than to the worst case (all components are affected). A weakness of the linear method lies in the time complexity when a subvector is inserted or removed into/from the global state vector. In analogy to Section 6.7.2, the worst case runtime is $O(m)$, even for a single insertion or removal. The method is still effective, if instructions that (de-)allocate memory regions are uncommon compared to instructions that merely change the contents of memory cells.

6.8.2 Balanced Tree Method

By introducing an additional data structure, we can reduce the asymptotic runtime to $O(\log m)$ in each case. For this purpose, we use a balanced e.g. AVL [AVL62] tree repre-

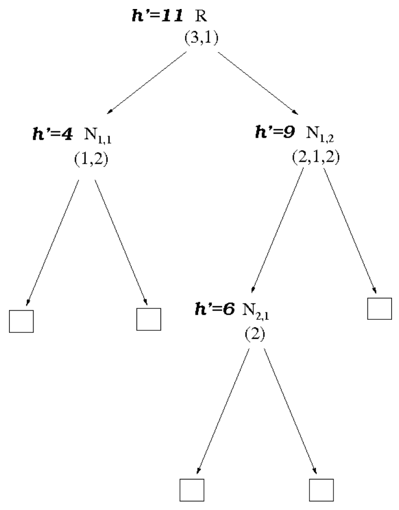

Figure 6.5: Example tree for calculating the hash code of a structured state vector

sentation t with m inner nodes - one for each subvector v_i, $i \in \{1, \dots, m\}$.

We define the hash function h' for node N in t as follows: If N is a leaf then $h'(N) = 0$. Otherwise,

$$h'(N) = h'(N_l) + h(v(N)) \cdot |\Sigma|^{|N_l|} + h'(N_r) \cdot |\Sigma|^{|N_l|+|v(N)|} \, mod \, q.$$

Here $|N|$ denotes the accumulated length of the vectors in a subtree with root N, and $v(N)$ stands for the subvector associated to N, while N_l, N_r denote the left and right subtrees of N respectively. If R is the root of t, then $h'(R)$ gives the same hash value as h applied to the concatenation of all subvectors in order. Figure 6.5 shows an example for a set of four vectors over $\Sigma = \{1, 2, 3\}$ and with $q = 17$. The nodes in the tree are annotated with their h' values, and are calculated as follows:

We obtain the same result (namely 11), for $h(1, 2, 3, 1, 2, 2, 1, 2)$ in the linear method.

When the values in a subvector v_i are changed, we must first re-calculate $h(v_i)$. Then the result must be propagated bottom-up to the root node. Because the depth of a balanced tree is bounded logarithmically in the number of nodes m, we traverse at most $O(\log m)$ nodes until we reach the root node of the tree. Furthermore, insertion and deletion of an element in/from a balanced tree can be performed in $O(\log m)$ time by restoring the balanced tree structure using rotations (either left or right). Updating the hash offsets for the two nodes on which a rotation is executed, is available in constant time. We summarize the observations in the following result.

Theorem 6.5. *Dynamic incremental hashing of a structured vector $v' = (v_1, \dots, v_m)$ with respect to its predecessor $v = (v_1, \dots, v_m)$ assuming a modification in component i (update within the component, insertion and deletion of the component), $i \in \{1, \dots, m\}$ with $I(v_i, v_i')$ being the set of indices that change in component i and $I^i_{\max} = \max_{(v,v')} |I(v_i, v_i')|$ is available in time*

- *$O(|I(v_i, v_i')|)$ for update, $O(m + \log n_i)$ for insertion, and $O(m)$ for deletion using $O(m + \max_{i=1}^m n_i)$ extra space.*

- $O(1)$ *for update,* $O(m + \log n_i)$ *for insertion, and* $O(m)$ *for deletion using* $O(m + \sum_{i=1}^{m}(n_i|\Sigma|)^{I^i_{\max}})$ *extra space.*
- $O(|I(v_i, v'_i)| \log m)$ *for update,* $O(\log m + \log n_i)$ *for insertion, and* $O(m)$ *for deletion using* $O(m + \max_{i=1}^{m} n_i)$ *extra space.*
- $O(\log m)$ *for update and* $O(\log m + \log n_i)$ *for insertion and* $O(\log m)$ *for deletion, using* $O(m + \sum_{i=1}^{m}(n_i|\Sigma|)^{I^i_{\max}})$ *extra space.*

Proof: For the linear method (first case) we have

$$\begin{aligned} h(v) &= \sum_{j=1}^{n_1} v_{i,j} \cdot |\Sigma|^j + \sum_{j=1}^{n_2} v_{2,j} \cdot |\Sigma|^{n_1+j} + \ldots + \sum_{j=1}^{n_k} v_{2,m} \cdot |\Sigma|^{\Sigma_{k=1}^{m-1} n_k + j} \, mod\, q \\ &= h(v_1) + h(v_2) \cdot |\Sigma|^{n_1} + \ldots + h(v_m) \cdot |\Sigma|^{\Sigma_{k=1}^{m-1} n_k} \, mod\, q, \end{aligned}$$

so that we can reduce the update to the static setting and multiply $|\Sigma|^{\Sigma_{k=1}^{i-1} n_k} \, mod\, q$ with the affected component. To update these additionally stored offset, we maintain a list $|\Sigma|^{n_j} \, mod\, q$, $j \in \{1, \ldots, m\}$. For insertion, $|\Sigma|^{n_i} \, mod\, q$ has to be newly computed. Using fast exponentiation, this value can be determined in time $O(\log n_i)$. Both tables consume $O(m)$ space in total. Insertion and deletion of a vector component additionally require $O(m)$ restructuring operations within the vector in the worst case.

For the balanced tree method (third case), insertion and deletion are available in time $O(\log m)$, while the update requires $O(|I(v_i, v'_i)| \log m)$ time. Once more, for inserting we additionally require $O(\log n_i)$ time for computing $|\Sigma|^{n_i} \, mod\, q$ from scratch. As a binary tree consumes space linear in the number of leaves, the space requirements are bounded by $O(m + \sum_{i=1}^{m} n_i)$.

As in the static case, improving factor $|I(v_i, v'_i)|$ to $O(1)$ (second and forth case) is available by precomputing $\sum_{j \in I(v_i, v'_i)}(-v_{i,j}|\Sigma|^j + v'_{i,j}|\Sigma|^j) \, mod\, q$ for all possible transitions from v_i to v'_i and $i \in \{1, \ldots, m\}$.

6.9 Incremental Hashing in StEAM

With the insights gained about the hashing of static as well as dynamic structured vectors, we have enough information to devise an incremental hash function for model checking C++ programs. A first implementation is available as an option in StEAM. The user should be given the choice whether to use partial, full or incremental hashing as each may be most appropriate depending on the checked program. If states are hashed explicitly, there is no risk that hash collisions may render the search incomplete and partial hashing may be preferable as resolving hash collisions is often faster than hashing the entire state (even if this were done incrementally). However, when bitstate-hashing or hash-compaction is used, hashing the full state vector is mandatory. Generally the incremental computation of the full hash function should be preferable, as we can expect it to be faster for most programs. In rare cases the state transitions of a program may affect a large part of the allocated memory. Due to the computational overhead of incremental hashing, it may be faster to use the full non-incremental hash computation in these cases. Experimental results on partial, full and incremental hashing follow in Chapter 8.

6.10 Related Work

6.10.1 Zobrist Keys

An early and frequently cited method for hashing are Zobirst Keys [Zob70]. Here, for a hash function $h : D^n \to E$, $|E| >> |D|$, each value from domain $d \in D$ is associated with a random number $r(d) \in E$. The hash code of $v = (v_1, \ldots, v_n) \in D^n$ is computed through the XOR operator as $\bigoplus_{i=1}^{n} r(v_i)$. Since the XOR operator is both, commutative and its own inverse, hash codes can be computed incrementally: For $v' = (v'_1, \ldots, v'_n)$, $v'_j = v_j$ $(j \neq i)$, it holds that $h(v') = h(v) \oplus r(v_i) \oplus r(v'_i)$. Zobrist keys are hence often cited in conjunction with board games, such as Chess [Mar91] or Go [Lan96]. As a clear advantage of the method, it is computationally inexpensive as it relies completely on XOR - in contrast to, e.g., Rabin-Karp hashing which requires multiplication and modulo operators. Through the commutativity of XOR however, all permutation of a sequence of values yield the same hash code, which is not the case for Rabin-Karp hashing.

6.10.2 Incremental Heap Canonicalization

In [MD05] the authors address incremental heap canonicalization (HC) in explicit state model checking. HC serves the purpose of obtaining a canonical representation (CR) for the heap of allocated memory regions, such that states which differ only in the addresses of the heap objects are detected as duplicates. R. Josif [Ios01] previously proposed an algorithm for HC, which concatenates the heap objects according to their depth-first order in the heap. The authors of the paper indicate two shortcomings of Iosifs algorithm: first, for each new state, the HC requires a full traversal of the heap, yielding a runtime linear in the number of heap objects. Second, according to the authors, small changes in the heap lead to major changes in the CR - i.e. the positions of many heap objects change. This can lead to a significant loss of speed in the exploration, if incremental hashing is used. In the paper, incremental hashing is described as maintaining separate hash values for each heap object, which get combined to a global hash value for the entire heap. After a state transition, the hash value is only recalculated for those objects that were either changed by the transition, or whose positions in the CR have changed. Since most transitions only change a small fraction of all heap objects, a heap HC which maintains the positions of most objects in the CR is advantageous. The paper devises an incremental HC, which solves the two shortcomings of Iosif's algorithm. First, the position for an object o in the CR is determined through the breadth-first access chain in the heap graph and the length of o. The breadth-first access chain is defined as the list of offset labels on the path edges. The authors devise a recursive relocation function, which computes an object's position in the CR according to its BFS access chain and length. The approach is implemented in the tools CMC and Zing. Experimental results for three example programs indicate a significant speedup of the exploration. In contrast to the work presented in this thesis, the paper misses to discuss the option of hashing only parts of the state, since when states are explicitly stored in the hash table, the exploration can be faster if the state is only partially hashed. Also, approximate hashing, such as bitstate-hashing is not mentioned although these techniques motivate incremental hashing in the first place through the need for a maximum degree of distribution. Also, the approach does not consider that for a state transition its is not only likely that a small subset of all heap objects are changed, but also that there are only small changes within those objects. The work in [MD05] requires, that the partial hash value of an object is fully recomputed, if it's contents have changed. This is surprising, since the hash function defined in the paper would also support incremental hashing within the single objects.

Chapter 7

State Reconstruction

The main problem that program model checkers face are high memory requirements, since for a complete enumeration the entire system state description has to be stored to allow pruning in case a state is visited twice. The storage problem as a consequence of the state explosion problem, is apparent in many enumeration attempts for model checking. The list of techniques is already long (cf. [CGP99]): *partial-order reduction* prunes the state space based on stuttering equivalence relations tracking commuting state transitions, *symmetry detection* that exploits problem regularities on the state vector, *binary state encodings* allows to represent larger sets of states, while performing a symbolic exploration with SAT formulae or BDDs, *abstraction methods* analyze smaller state spaces inferred by suitable approximation functions, and *bit-state hashing* and *hash compaction* compress the state vector down to a few bits, while failing to disambiguate some of the synonyms.

Here we add a new option to this list, which will cooperate positively with explicit-state heuristic search enumeration. In *state reconstruction* for each state encountered either as a state to be expanded or as a synonym in the hash table, we reconstruct the corresponding state description via its generating path. In a first phase predecessor links are used to reconstruct the path, while in the second phase the path is simulated to reconstruct the state starting from the initial one. This reduces the actual state description length down to a constant amount. The approach is feasible, if the simulation mode that reconstructs a state based on its generation path is fast. For improved differentiation in duplicate detection we may include further information with each state.

7.1 Memory Limited Search

To alleviate the storage problem for exploration algorithms, in *Artificial Intelligence* different suggestions have been contributed. *Iterative deepening A* search* (cf. Section 3.3.3) refers to a series of depth-first searches with increasing depth or cost threshold. Transposition (hash) tables [RM94] include duplicate detection to depth-first (iterative-deepening) search. Memory-limited algorithms generate one successor at a time. The algorithms assumes a fixed upper limit on the number of allocated edges in $open \cup closed$. When this limit is reached, space is reassigned by dynamically deleting one previously

expanded node at a time, and if necessary moving its parent back to *open* such that it can be regenerated later. In acyclic state spaces *Sparse Memory Graph Search* takes a compressed representation of the *closed* list that allows the removal of many, but not all nodes. *Frontier Search* [KZ00] completely omits list *closed* and applies a divide-and-conquer strategy to find the optimal path in weighted acyclic graphs. *Partial Expansion* A* is an algorithm that reduces the *open* list. It contributes to the observation that in classical A* search that expands a node by always generating *all* of its successors; therefore, nodes with merit larger than the optimal solution which will never be selected, which can clutter the *open* list, and waste space. In *FSM pruning* [TK93] the hash table is replaced by an automatically inferred structure based on regularities of the search space. *Pattern Databases* (cf. Section 6.5.4) are memory-intense containers to improve the heuristic estimate for directed search and refer to an approximation that is a simulation. *Delayed duplicate detection* [Kor03] refers to the option to include secondary memory in the search.
For assembly-level model checking of software we observed that duplicate detection for the large state descriptions is essential, for full verification lossless storage is required, there is no general assumption on the regularity of the state space graph to be made, and, considering compacted atomic code regions as in our case, the graph may be weighted. Therefore, the application of most of the above techniques is limited.

In [EPP05], the authors propose a memory-efficient representation of markings in colored petri nets. The approach distinguished between *explicit-makings* and Δ-markings. Explicit markings, which only appear at levels $i \cdot n$, $n \in I\!N$, $i = 1, 2, ..$ store the full information about a marking, while Δ-markings are represented through a reference to one of it's predecessors and a transition instance. We will outline the difference to our approach in Chapter 9.

7.2 State Reconstruction in StEAM

Our proposal of *state reconstruction* distributes the state by its hash value, but stores only limited information that is used to reconstruct the state. The general assumption is given a sequence of instructions, reconstruction is fast.

Mini-States For state reconstruction, we introduce the notion of mini-states, which constitute a constant-sized, compressed representation of actual states in StEAM. A mini-state is essentially composed of:

- The pointer to the immediate (mini-state) predecessor to reconstruct the generating path.
- The information, which object code transition was chosen from the parent state to the current one.
- Additional hash information for the corresponding explicit state.

The object code transition is derived from the program counter of one thread in the explicit state, which the mini-state corresponds to. Storing the additional hash information in a mini state is not mandatory, as it can merely save the reconstruction of states in case of a hash conflict. However, since state comparisons are computationally expensive, it is most advisable to invest an additional constant amount of memory per state to store the additional information, as it may greatly speed up the exploration.

Figure 7.1 illustrates the method of state reconstruction. On the upper side of the figure, we see the first three levels of the search tree with a degree of three. Assuming a worst-case scenario, we need to store in each node, the full description of the corresponding state, including stack contents, CPU-registers, global variables and the lock- and memory pool. The root node (i.e. the initial state) is labeled by I. The other nodes are labeled by $1,..,12$ according to their breadth-first generation order. The edges of the tree are labeled with o_i - which denotes the object code instruction (operator) generating state i when applied to i's immediate predecessor state. The edges are directed towards each state's predecessor to emphasize that the search tree in StEAM is build by placing a predecessor pointer in each state.

As there is theoretically no limit to the amount of dynamically allocated memory in a program, the required memory for explicitly storing the generated states in the search tree can grow to an arbitrary amount.

On the lower side of Figure 7.1 we see the same search tree encoded with mini-states. Besides the predecessor pointer and the operator, which are represented by the labeled edges, the only information stored in a mini state is the additional hash code h_i of state i.

In addition to the mini-states, we also have to memorize I as the only explicit state. In Figure 7.1, state 5 and its corresponding mini-state are highlighted by a bold dashed border. To reconstruct state 5 from the mini-state encoding, we follow the predecessor pointers up to the (mini-) root node, which yields the operator sequence o_5, o_1. Applying the converse sequence o_1, o_5 to I, transforms I to state 1 and finally to state 5. Algorithm 7.1 illustrates the expansion of a mini-state state s using reconstruction. Here, I describes the initial state in its full representation and I_{min} the corresponding mini-state. For a mini-state s, $s.o$ denotes the operation (e.g. the sequence of machine instructions), which transforms the predecessor $s.pred$ into s. Similarly, for a full state x, $x.code(x')$ denotes the operation which transforms x to it's successor state x'. Note, that StEAM operations have a constant-sized representation through their program counter, which is usually the program counter of a thread running in x. The notion $x.execute(o)$ refers to applying operation o to a full state x. The hash table H contains the mini-state representatives of all previously generates states.

State reconstruction applies to both lists *open* and *closed*. In case a duplicate candidate is found by a matching hash-value and additional mini-state information, the corresponding state is reconstructed by simulation of the implicit trail information from a copy of the initial state. Since as in our case, virtual machines are highly tuned for efficiency of linear program execution, reconstruction is fast. *State reconstruction* can be combined with almost all other state space reduction techniques.

A theoretical limit to state reconstruction is the lower bound for a lossless storage of states, which is about $O(\log \binom{n}{m})$, where n is the number of elements to be stored and m is number of bits available for the hash table. By the discrepancy of the number of states generated and the state description needed, the practical savings are large.

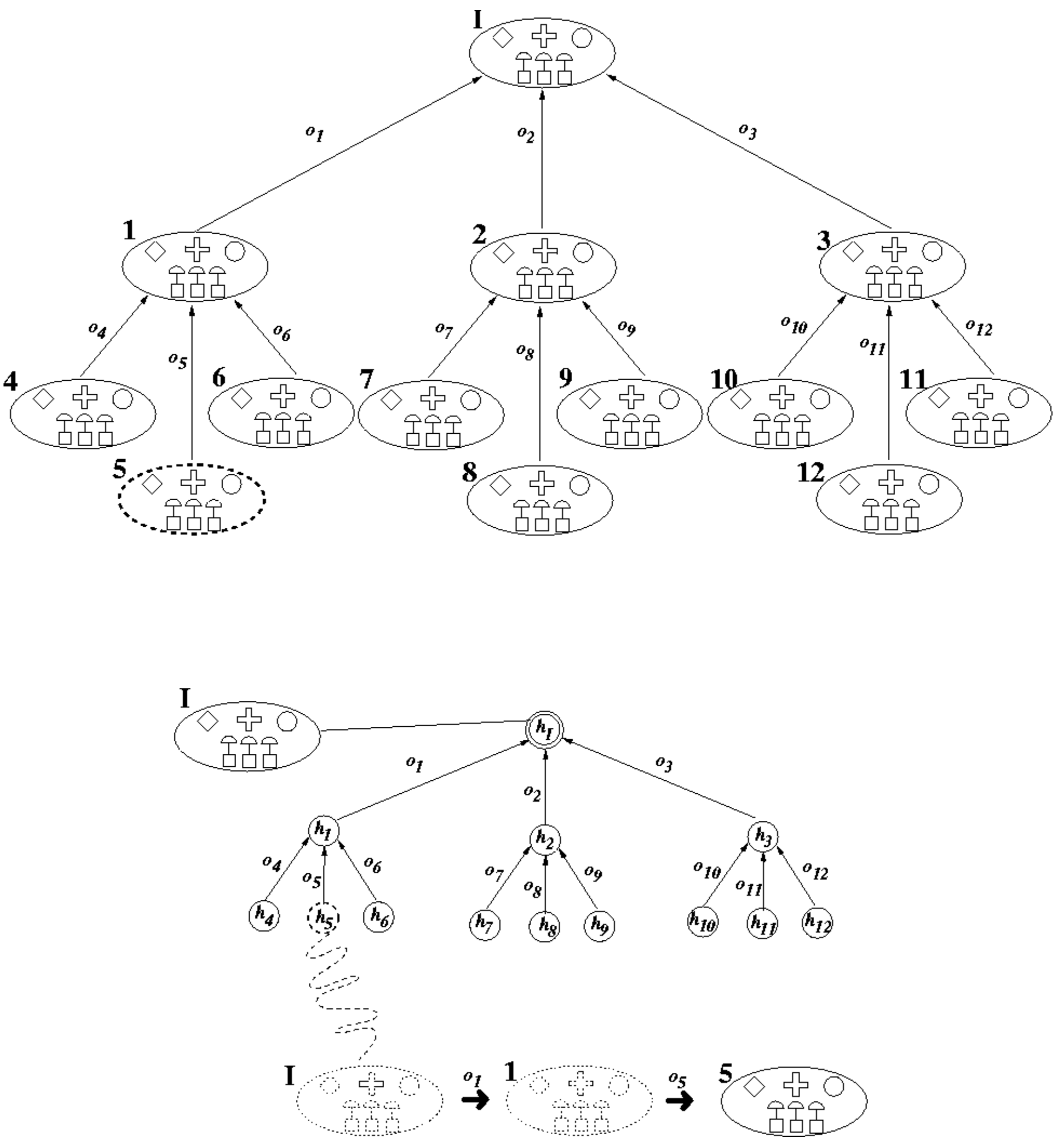
I
o_1 o_2 o_3
1 2 3
o_4 o_5 o_6 o_7 o_8 o_9 o_{10} o_{11} o_{12}
4 5 6 7 8 9 10 11 12
I
h_I
o_1 o_2 o_3
h_1 h_2 h_3
o_4 o_5 o_6 o_7 o_8 o_9 o_{10} o_{11} o_{12}
h_4 h_5 h_6 h_7 h_8 h_9 h_{10} h_{11} h_{12}
I o_1 1 o_5 5

Figure 7.1: Search tree in explicit representation (top) and using mini-states (bottom).

```
ExpandRec(mini-state $s$)
    $x \leftarrow Reconstruct(s)$
    $\Gamma \leftarrow expand(x)$
    for each $x' \in \Gamma$
        $c \leftarrow hash(x')$
        $dup \leftarrow false$
        for each $s' \in H[c]$
            if (Reconstruct($s'$)=$x'$)
            $dup \leftarrow true$
            break
        if ($dup = false$)
            <allocate mini-state $s''$>
            $s''.pred \leftarrow s$
            $s''.o \leftarrow x.code(x')$
            $H[c].store(s'')$
```

```
Reconstruct(mini-state $s$)
    stack $\sigma \leftarrow \emptyset$
    $s' \leftarrow s$
    while ($s' \neq I_{min}$)
        $\sigma.push(s'.o)$
        $s' \leftarrow s'.pred$
    $x \leftarrow I$
    while ($\sigma \neq \emptyset$)
        $o \leftarrow \sigma.pop()$
        $x.execute(o)$
    return $x$
```

Algorithm 7.1: General State Expansion Algorithm with State Reconstruction

As other areas related to state space search, such as puzzle solving or planning, have a low memory requirement of a few bytes per state, regeneration would possibly not yield gain. The larger the state description length, the better the obtained savings.

Experimental results for state reconstruction can be found in Chapter 8.

7.3 Key Sates

As mentioned, due to the high speed of the virtual machine, state reconstruction is generally fast in StEAM. However, the approach may also considerably slow down the exploration. This happens in cases, where large search depths are encountered or when there are a lot of hash collisions: for each stored sate x' with the same hash address as the newly generated state x, to check for a duplicate, we must reconstruct x' along the lengthy path from the root node to the mini-state of x' (cf. Algorithm 7.1). In an advanced application of state reconstruction, we may introduce so-called *key-states*. The latter are full descriptions of states, that for $i \in I\!N$ are associated to the mini-states at levels $i \cdot j$, $j = 1, 2, ...$ When we want to reconstruct the full description of some mini-state s, we do not have to follow the predecessor pointers up to the root of the search tree, but merely to the next predecessor with an associated key state. This ensures, that we need to execute at most $i - 1$ operations to reconstruct a state. Figure 7.2 illustrates a search tree for $i = 3$.

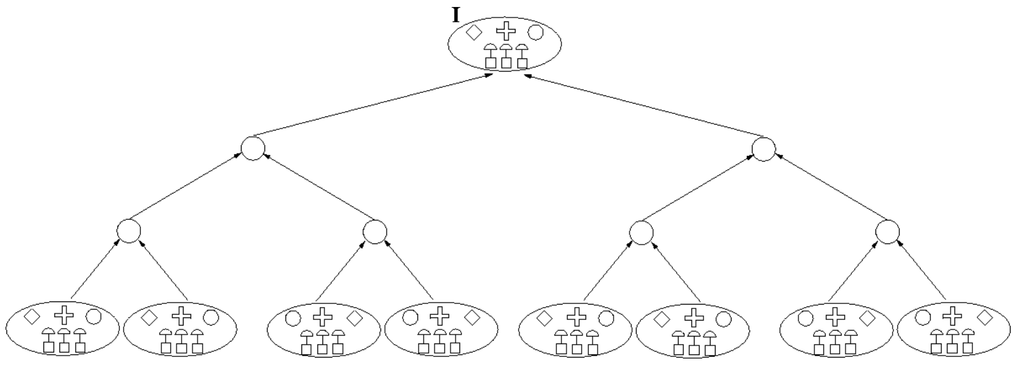

Figure 7.2: Search tree with key states.

7.4 Summary

In summary, state reconstruction contributes a new approach for improving the time-space trade-off in program model checking in recomputing system states based on their generating paths. While it is lossless in general and combines with any state expanding exploration algorithm, bit-state hashing can be integrated for the *closed* list. The main aspect is that no longer system states are stored but reconstructed through predecessor information. The memory requirement per state shrinks to a constant and the time offset is affordable (cf. Chapter 8). In fact larger models can be analyzed. In the current approach, each state in the search tree is reconstructed starting from the root node. Thus we have low memory requirements, since only the initial system state must be kept in memory in a full representation. As a drawback, the reconstruction of a state can be time consuming, if its generating path is long. One issue for future research is the integration of key-states into the existing implementation.

Chapter 8

Experiments

In the following, we experimentally evaluate the approaches presented in the previous chapters.

First, we compare heuristic search with undirected search in StEAM. Then, it is evaluated, in how far state reconstruction (cf. Chapter 7) can help to alleviate the memory requirements of software model checking, and thus to find errors in larger programs. In Section 8.4, we investigate the speedup in the state exploration when incremental hashing is used. Finally, in Section 8.5, we evaluate the planning capabilities of StEAM for the multi-agent manufacturing problem (cf. Section 5.5).

8.1 Example Programs

Here, we discuss some programs used for benchmarking in StEAM. The programs include C++ implementations of classical toy protocols, as well as simple real-life applications. Moreover, the examples are either scalable or non-scalable programs and contain different types of errors, such as deadlocks, assertion- and privilege-violations.

8.1.1 Choosing the Right Examples

Classical model checking usually investigates models of concurrent systems described in specialized modeling languages such as Promela. In contrast, unabstracted program model checking has a broader spectrum of applications, as we are not limited to the search for fundamental errors - such as errors in protocol specifications. The new generation of program model checkers also allow us to find low level programming errors, like illegal memory accesses. On the other hand it is good to point out, how program model checking is related to the classical applications. In particular, it should be evident that the used description languages (i.e. actual programming languages), subsume the specialized languages of classical model checkers with respect to expressiveness.

For the above reasons, it is reasonable to include in the set of test programs implementations of protocols that are frequently cited in the classical model checking research. Additionally, we should exemplify how the tool can be used to find errors during the implementation phase of software. Hence, we implemented the popular protocols: *Dining*

Philosophers, *Leader Election* and *Optical Telegraph* as concurrent C++ programs. Additionally, we devised a bank automata scenario whose implementation was constantly checked for errors.

8.1.2 Scalability

A scalable model (or program) is one that is instantiated through a size parameter n, which usually indicates the number of involved processes (or threads). It is a known fact, that state space of a system grows exponentially with the number of involved processes. Hence, the quality of model checking techniques is often measured by the maximal scale parameter in a model, for which the an error is found. However, not for all systems, the difficulty to find an error grows monotonic with the number of processes. As a simple example, consider a scalable version of the account balance example from section 4.5.2: We have n processes, which access a single shared variable x. A data inconsistency arises, if another process accesses x, before a process has written the results of its algebraic operations back to x. For this example, the error probability even increases with the number of processes.

8.1.3 Dining Philosophers

The dining philosophers problem constitute an popular scalable toy protocol. It is described as n philosophers sitting on a round table. Each philosopher takes turns in philosophizing and eating. For the latter, there is a plate of spaghetti in front of each philosopher. Between two philosophers, lies a fork, i.e. n forks in total. In order to eat, a philosophers needs to hold a fork in each hand. So - when he feels hungry, he stops philosophizing and first takes the fork to his left and then the fork to his right. When finished eating, the philosopher will put down the forks and continue to philosophize. When a philosopher tries to take a fork that is already used by another philosopher, he will wait until it is put down. In a faulty protocol, a deadlock occurs, if all philosophers pick up the left fork at the same time, because then each one, waits for the second fork to be released by his right neighbor. As a consequence, the philosophers will starve to death. Figure 8.1 illustrates the initial state for $n = 4$.

Although often addressed as a toy example, the dining philosophers play an important role in protocol verification, and not only so because it is a clear and funny way to illustrate deadlocks. Through it's scalability, the protocols state space can be increased to an arbitrary size. Furthermore, while the size of the state space increases exponentially with the number of processes, the shortest path from the initial state to a deadlock grows only linearly and can be extrapolated. This qualifies the protocol as a way to test the quality of heuristics.

8.1.4 Optical Telegraph

This protocol involves a set of n communicating optical telegraph stations [Hol91]. The stations form a chain to transmit messages over long distances. When a station relays a message, it first requests the attention of its neighbor station then it sends the message.

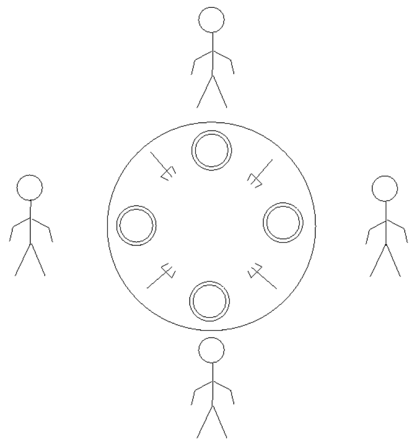

Figure 8.1: The Dining Philosophers.

A deadlock occurs, if all stations mutually request each other's attention. Though still a toy program, an implementation of the optical telegraph protocol gets closer to a real application than the dining philosophers. It is also frequently cited in model checking related literature [GV02, MJ05, LV01, Meh02, LL03].

8.1.5 Leader Election

A leader election problem involves a set of n processes connected through communication channels in an unidirectional ring. Figure 8.2 illustrates such a scenario.

The task is to elect one process as the leader by message passing. Hence, a necessary criterion for a valid leader election protocol is that at each time the number of elected leaders is at most one. Such a valid protocol problem is given e.g. in [FGK97]. An error can easily be seeded into a valid implementation, by assigning the same id to more than one process. As a result, more than one leader can be elected.

Observer Threads Obviously, the above criterion is a safety property that must be expressed as an invariant over the entire program execution. As stated in Section 4.4, StEAM currently supports the automatic detection of privilege violations, deadlocks and illegal memory accesses and allows the definition of local assertions. We can augment the checked properties to include invariants by introducing *observer threads*. These threads then constitute a part of the model checking environment rather than the actual program. Concretely, for an invariant demanding that property ϕ must hold at any time, we define a thread which executes *VASSERT*(ϕ) in an infinite loop. As a result, for each state in the original program, we have a state in the new program which differs only in the thread-state of the observer. Furthermore, for each such state, there is a successor state in which the assertion is is checked.

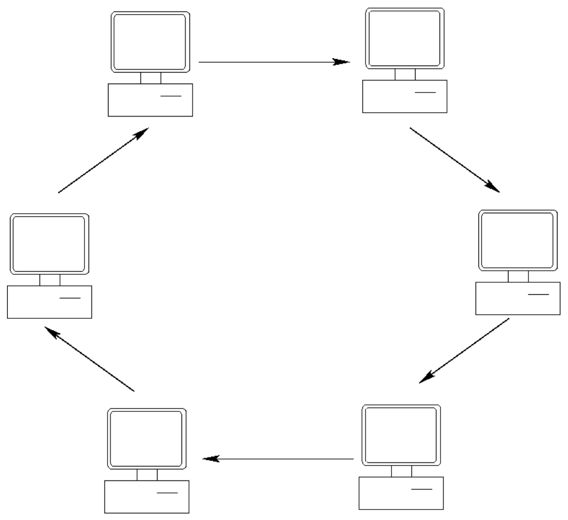

Figure 8.2: Example for a leader election problem.

8.1.6 CashIT

CashIT describes a scenario, where the controllers for n cash dispensers communicate with a global database of bank accounts through communication channels (cf. figure 8.3). After logging in, the users of the automata may deposit, withdraw or transfer money from/to the accounts.

CashIT differs from the first three examples, not only because it was devised specifically for the use with StEAM. Also, the example was written from a different perspective. For toy models such as the dining philosophers, we a priori know their implementations and the errors they yield.

For CashIT it was tested, in how far StEAM can be used to detect errors during the implementation phase of software, rather than during maintainance. Hence, while writing the program, the partial implementation was regularly tested against several properties, including (besides the absence of deadlocks and privilege violations):

- For each withdrawal, the account balance decreases by the withdrawn amount.
- Analogously, for deposits the balance increases.
- For for a transfer of n units from account a to account a', the balance of a decreases by n, while the balance of a' increases by n.

Although, the above properties may sound trivial, their verification is not. In fact, model checking with StEAM revealed several errors that arose during the implementation of CashIT. The most subtle of which, is a privilege violation that, according to the error trail, occurred during withdrawal. As it turns out, the error was caused by a false lock to the account balance:

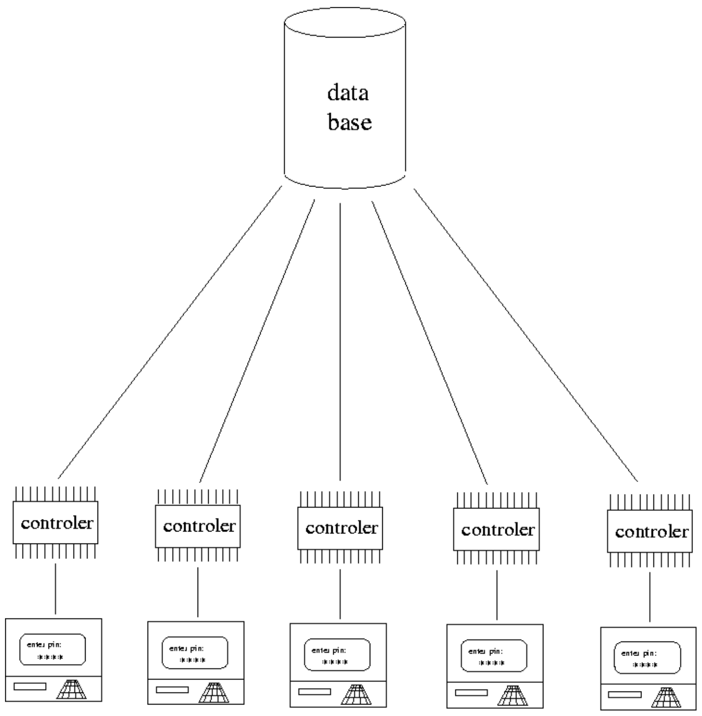

Figure 8.3: The CashIT program.

```
VLOCK(balances[logged_user])
```

which should have been:

```
VLOCK(&balances[logged_user])
```

As the CashIT example indicates, the current version of StEAM is already useful to detect program errors during the implementation phase of a simple program.

8.2 Experiments on Heuristic Search

Here, we compare the results of uninformed search in StEAM with those obtained through directed search with the heuristics explained in Section 4.5. The machine used is a Linux PC with a 3GHz CPU and 1GB of RAM.

We conducted experiments with the four scalable programs from Section 8.1, dining philosophers problem (*philo*), the leader election protocol (*leader*), the optical telegraph protocol (*opttel*), and the bank automata scenario (*cashit*).

Lock and Global Compaction Lock and Global Compaction (lgc) is a state space reduction technique that was devised for StEAM. It alleviates the state explosion problem by allowing thread switches only after a resource has been locked or unlocked, or after an access to the memory cell of a global variable has occurred. The intuition behind lgc is, that only global variables and resource locks can influence the behavior of a thread: Assume that at some state s, the program counter of thread t is at a conditional branch. Furthermore by executing another thread t' in s, we arrive at s'. Obviously, the state resulting from iterating t in s' can only differ from the state resulting from iterating t in s, if the iteration of t' in s changes the value of some global variable which occurs in the branch condition of t.

Analogously, if the next action of t is to lock a resource r, the iteration of t' can only influence t, if it locks or unlocks r.

In the the experiments on directed and undirected search, it is determined in how far lgc can help to solve greater program instances.

Undirected Search The undirected search algorithms considered are BFS and DFS. Table 8.1 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 seconds 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 insignificantly 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 8.2 and Table 8.3 show the results for best-first search applied to the four programs. Analogously Table 8.4 and Table 8.5 depict the results for A*.

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*, lead 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

		cashit				leader			
	c	s	l	t	m	s	l	t	m
DFS	y	1	97	15	134	80	1,197	25	739
DFS	n	1	1,824	2	96	12	18,106	10	390
BFS	y	-	-	-		5	56	36	787
BFS	n	-	-	-		3	66	3	146

		opttel				philo			
	c	s	l	t	m	s	l	t	m
DFS	y	9	8,264	21	618	90	1,706	17	835
DFS	n	6	10,455	22	470	20	14,799	46	824
BFS	y	2	21	2	97	6	13	16	453
BFS	n	-	-	-		4	26	6	164

Table 8.1: Results with Undirected Search.

		cashit				leader			
SA	C	s	l	t	m	s	l	t	m
pl1	y	1	97	15	134	5	73	9	285
pl2	y	1	97	6	120	5	73	9	312
mb	y	1	97	15	134	80	1,197	25	737
lnb	y	1	97	15	134	5	73	9	263
int	y	1	98	9	128	5	57	39	789
aa	y	1	84	0	97	5	72	28	637
rw	y	1	61	0	97	60	661	19	671
pba	y	1	97	9	121	80	1,197	24	740
pbb	y	1	97	15	134	80	1,197	24	741
pl1	n	1	1,824	2	97	4	245	9	284
pl2	n	1	792	0	97	4	245	17	376
mb	n	1	1,824	2	97	12	18,106	10	428
lnb	n	1	1824	2	97	4	245	9	284
int	n	1	1,824	2	97	3	68	4	139
aa	n	2	328	9	146	3	132	3	139
rw	n	2	663	15	463	40	1,447	10	470
pba	n	1	792	1	97	12	18,106	11	411
pbb	n	1	1,824	2	97	12	18,106	10	407

Table 8.2: Results with Best-First Search for cashIT and Leader Election.

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 Figures 8.4, 8.5, 8.6 and 8.7 give a graphical summary of the impact of heuristic search on the overall performance of the model checker. We measure the performance by extracting the fourth root of the product of trail length, processed states, time and memory (geometric mean of the arguments). We use a logarithmic scale on both axes. Note, that there is no figure for cashIT, simply because the available experimental data is insufficient for a graphical depiction. Also, except for Leader Election, we only give the diagram for *lgc*, since as it generally performs better. For Leader Election, we additionally give a diagram for the absence of *lgc*, because here the difference

		opttel				philo			
SA	C	s	l	t	m	s	l	t	m
pl1	y	60	602	21	770	190	386	17	622
pl2	y	60	601	21	680	190	385	49	859
mb	y	9	6,596	16	518	150	750	21	862
lnb	y	60	602	21	763	190	386	17	639
int	y	2	22	1	98	8	18	19	807
aa	y	-	-	-		6	14	21	449
rw	y	2	309	1	98	90	1,706	18	854
pba	y	9	6,596	16	513	90	450	4	234
pbb	y	9	6,596	16	510	150	750	21	860
pl1	n	40	1,203	20	705	150	909	34	859
pl2	n	40	1,202	19	751	150	908	33	857
mb	n	6	10,775	23	489	60	1,425	10	548
lnb	n	40	1,203	18	719	150	909	34	856
int	n	-	-	-		5	33	23	663
aa	n	-	-	-		4	27	7	197
rw	n	2	494	7	206	20	14,799	469	822
pba	n	6	10,775	23	470	40	945	3	239
pbb	n	6	10,775	24	471	60	1,425	10	571

Table 8.3: Results with Best-First Search for Optical Telegraph and Dining Philosophers.

		cashit				leader			
	c	*s*	*l*	*t*	*m*	*s*	*l*	*t*	*m*
pl1	y	-	-	-		5	56	34	711
pl2	y	1	61	278	387	5	56	36	760
mb	y	-	-	-		5	56	40	769
lnb	y	-	-	-		5	56	34	719
int	y	-	-	-		5	57	40	785
aa	y	-	-	-		5	61	38	744
rw	y	1	61	196	310	7	78	16	589
pba	y	-	-	-		5	56	38	648
pbb	y	-	-	-		5	56	40	777
pl2	n	1	136	1057	614	3	66	3	141
mb	n	-	-	-		3	66	3	151
rw	n	-	-	-		3	66	2	97
pba	n	-	-	-		3	66	3	138
pbb	n	-	-	-		3	66	3	139

Table 8.4: Results for A* on cashIT and Leader Election.

		opttel				philo			
	c	*s*	*l*	*t*	*m*	*s*	*l*	*t*	*m*
pl1	y	3	32	4	215	190	386	17	631
pl2	y	2	21	0	98	190	385	48	860
mb	y	2	22	2	98	7	16	11	371
lnb	y	2	22	1	98	8	18	8	286
int	y	2	22	1	98	7	16	18	629
aa	y	2	22	4	171	6	14	21	449
rw	y	2	22	2	98	6	14	21	449
pba	y	3	32	13	424	60	122	19	628
pbb	y	3	32	19	573	60	122	19	634
pl2	n	-	-	-		5	38	17	485
mb	n	-	-	-		4	27	4	155
rw	n	-	-	-		4	27	7	199
pba	n	2	63	55	845	7	51	42	872
pbb	n	-	-	-		6	45	19	431

Table 8.5: Results for A* for Optical Telegraph and Dining Philosophers.

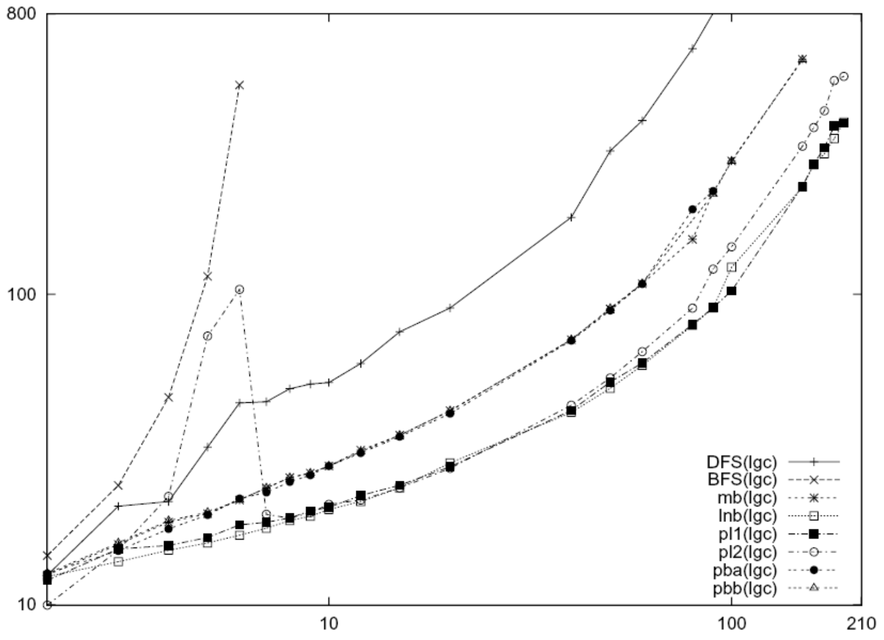

Figure 8.4: Geometric mean for Dining Philosophers with lgc.

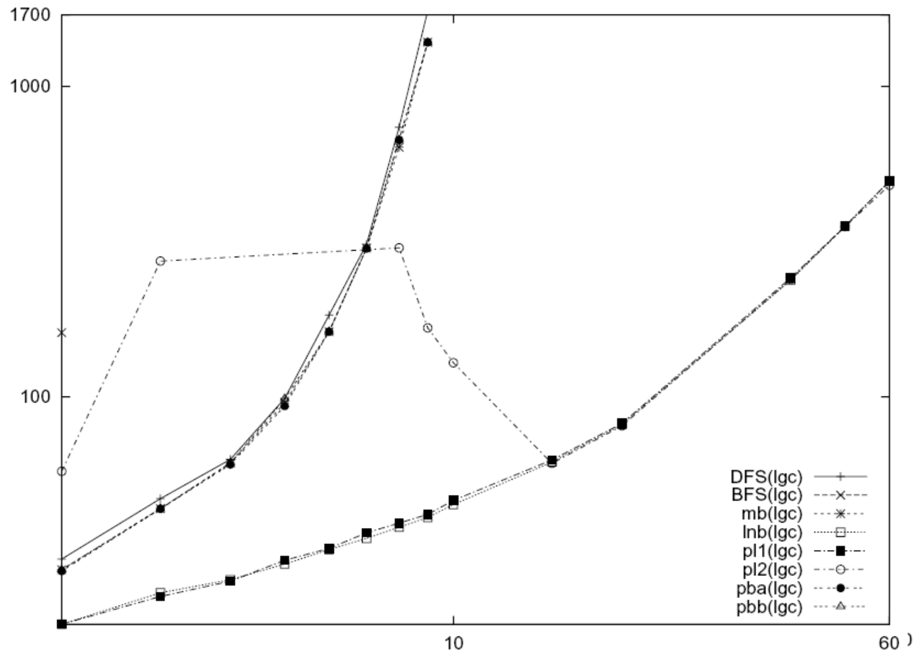

Figure 8.5: Geometric mean for Optical Telegraph with lgc.

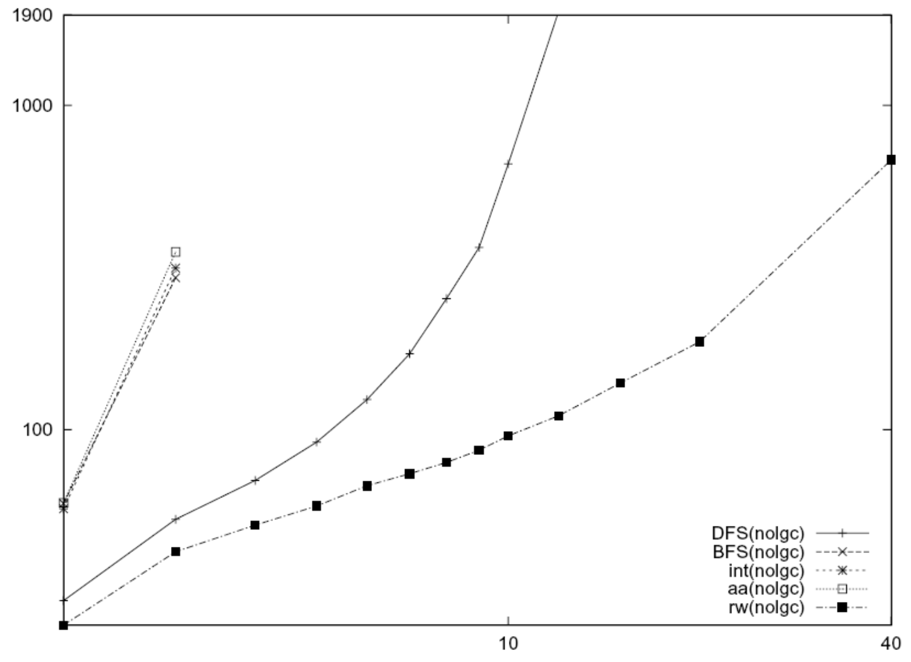

Figure 8.6: Geometric mean for Leader Election without lgc.

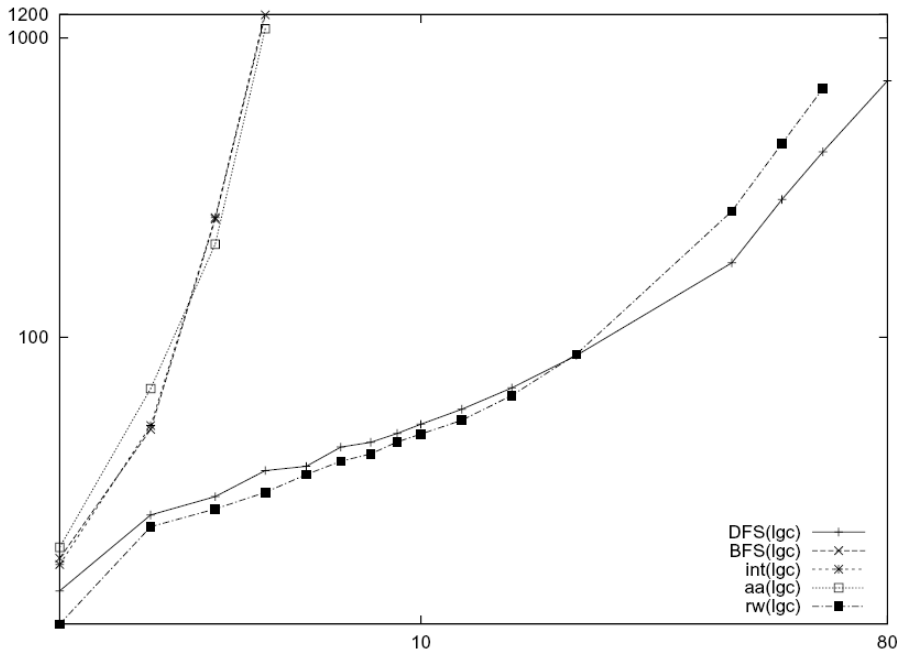

Figure 8.7: Geometric mean for Leader Election with lgc.

to the setting with *lgc* is most significant. In fact, the maximum possible scale almost doubles, if the compaction is used.

In Optical Telegraph, 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.

In the Dining Philosophers, 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*.

8.3 Experiments for State Reconstruction

To validate the memory savings of state reconstruction, we used a Linux system with a 1.8 GHz CPU and a memory limit of 512 MB and compare the runtime in seconds and space in MB. The time limit was set to 30 minutes.

We ran *StEAM* with (rec) and without (¬rec) state reconstruction in the algorithm breadth-first search (BFS) and greedy best-first search (GBF) with the 'lock and block' [LME04] heuristic. The models analyzed are instances of the Dining Philosopher (phil) and the Optical Telegraph protocol (optel). The results are summarized in Table 8.6 and 8.7. As we can see the approach trades time for space. This tradeoff becomes more significant while scaling the model. When using 'rec' in heuristic search, we are even able to find errors in instances, where ¬rec fails.

In fact, when for GBF and 'rec' the search eventually exceeds the time limit in optel-47, the memory consumption is still low (110 MB). So it can be assumed, that even higher instances can be handled, if more time is available.

	BFS				GBF			
	¬rec		rec		¬rec		rec	
model	time	space	time	space	time	space	time	space
phil-2	0	97	0	97	0	97	0	97
phil-3	0	101	1	98	0	97	0	97
phil-4	4	133	27	108	0	97	0	97
phil-5	29	402	808	275	0	98	0	97
phil-10	o.m.	o.m.	o.m.	o.m.	0	98	56	97
phil-100	o.m.	o.m.	o.m.	o.m.	5	199	211	100
phil-190	o.m.	o.m.	o.m.	o.m.	444	492	663	107
phil-195	o.m.	o.m.	o.m.	o.m.	o.m.	o.m.	1,540	108
phil-196	o.m.	o.m.	o.m.	o.m.	o.m.	o.m.	1,557	108

Table 8.6: Results of State Reconstruction for the Dining Philosophers.

	BFS				GBF			
	¬rec		rec		¬rec		rec	
optel-2	36	271	996	185	0	98	0	97
optel-5	o.m.	o.m.	o.t.	o.t.	0	102	0	98
optel-10	o.m.	o.m.	o.t.	o.t.	1	116	3	98
optel-20	o.m.	o.m.	o.t.	o.t.	15	183	38	100
optel-40	o.m.	o.m.	o.t.	o.t.	137	511	850	106
optel-41	o.m.	o.m.	o.t.	o.t.	o.m.	o.m.	980	107
optel-46	o.m.	o.m.	o.t.	o.t.	o.m.	o.m.	1,714	110
optel-47	o.m.	o.m.	o.t.	o.t.	o.m.	o.m.	o.t.	o.t.

Table 8.7: Results of State Reconstruction for Leader Election.

8.4 Experiments on Hashing

Here, we experimentally measure the impact of incremental hashing on the exploration speed of StEAM.

Partial vs Full Hashing For the previous experiments, we used *partial hashing*, i.e. for efficiency considerations, only a part of the state description is hashed. In this case, we only hash the machine registers of all running threads. This allows us to solve higher instances of each protocol within the given time limit. Table 8.8 shows the number of hash collisions which occur when partial hashing is used compared to the number of collisions using *full hashing* i.e. if the full state vector is hashed. Here, we use the dining philosophers with depth-first search.

model	states	full	partial
phil-2	58	0	6
phil-3	231	0	23
phil-4	969	0	135
phil-5	2,455	0	304
phil-10	18,525	4	1,656
phil-15	44,356	19	3,470
phil-20	80,006	47	5,746

Table 8.8: Number of hash collisions for partial and full hashing.

Although the machine registers are the state components most likely to be changed by a transition, partial hashing produces a much greater number of hash collisions than full hashing. For a scale of 20 and 80,006 generated states, we have 47 as compared to 5746 collisions. This is not a big problem for the small test programs used here: It turns out that in none of the experiments, a hash code occurs more than twice, since the size of the hash table is relatively big compared to the number of generated states. Hence, a hash collision can quickly be resolved. However, with bitstate hashing in mind, the amount of collisions would be unacceptable: each collision leads to an unexplored part of the search tree, increasing the probability of missing an error. This implies, that for bitstate hashing it is essential to have a minimum of hash collisions, which is only possible if the entire state description is hashed. This also includes the stacks and all allocated memory blocks. As a consequence, computing the hash address may considerably slow down the exploration.

Incremental Hashing We conducted some first experiments on incremental hashing of the stacks in StEAM. The rest of the state is hashed non-incrementally and the result are combined to a global hash value. For each running thread the program stack is given as a byte vector of 8 MByte, while only the used parts are stored in the state description. By concatenating the stack of n running threads, we arrive at a vector size of $8 \cdot 1024^2 \cdot n$, which has to be hashed. As new threads may be generated, the vector can grow dynamically. We used the same machine, as well as the same time- and memory limits as in the preceding experiments. Figure 8.8 compares the run times of the exploration with non-incremental and incremental hash computation for the dining philosophers, optical telegraph and leader election protocol[1]. For the leader election protocol with GBF search, we use the more appropriate 'read/write' [LME04] heuristic.

Note, that even for the non-incremental computation, we only regard the portion of the stack, which lies below the current stack pointer. All memory cells above the stack pointer are treated as zero. This is important, since hashing the unused portions of the stack in the non-incremental case would not yield a fair comparison and a considerable speedup of the exploration could be taken for granted.

The results are encouraging, as we get a speedup by factor 10 and above compared to non-incremental hashing and more errors can be found within the time limit. In the near future we aim at the verification of more complex programs, that will not allow to store each state explicitly. This implies that compacting hash schemes such as bitstate hashing are required. As stated before, partial hashing is not appropriate to hash compaction due to the high number of hash collisions. Hence, even though partial hashing is faster for the small test programs, full hashing will be essential for checking larger programs for which our incremental hashing scheme provides an efficient way.

[1]The leader election protocol makes use of the nondeterministic RANGE-statement, which is currently not supported for state reconstruction. Hence, it was not used in the preceding experiments.

	BFS		GBF	
model	¬inc	inc	¬inc	inc
phil-2	1	0	0	0
phil-3	14	1	0	0
phil-4	257	25	0	0
phil-5	o.t.	o.t.	1	0
phil-6	o.t.	o.t.	1	0
phil-7	o.t.	o.t.	1	0
phil-8	o.t.	o.t.	2	0
phil-9	o.t.	o.t.	2	0
phil-10	o.t.	o.t.	3	0
phil-16	o.t.	o.t.	10	0
phil-17	o.t.	o.t.	13	1
phil-18	o.t.	o.t.	14	1
phil-19	o.t.	o.t.	17	1
phil-20	o.t.	o.t.	20	1
phil-25	o.t.	o.t.	37	3
phil-30	o.t.	o.t.	65	4
phil-50	o.t.	o.t.	279	19
phil-100	o.t.	o.t.	o.t.	239

	BFS		GBF	
model	¬inc	inc	¬inc	inc
optel-2	o.t.	o.t.	1	0
optel-3	o.t.	o.t	3	0
optel-4	o.t.	o.t.	6	0
optel-5	o.t.	o.t.	11	0
optel-10	o.t.	o.t.	79	6
optel-11	o.t.	o.t.	105	7
optel-12	o.t.	o.t.	135	9
optel-13	o.t.	o.t.	170	12
optel-14	o.t.	o.t.	214	15
optel-15	o.t.	o.t.	262	19
optel-16	o.t.	o.t.	316	23
optel-17	o.t.	o.t.	379	27
optel-18	o.t.	o.t.	448	32
optel-19	o.t.	o.t.	524	39
optel-20	o.t.	o.t.	612	46
optel-25	o.t.	o.t.	1,187	113
optel-30	o.t.	o.t.	o.t.	257
optel-40	o.t.	o.t.	o.t.	o.t.

	BFS		GBF	
model	¬inc	inc	¬inc	inc
leader-2	6	0	0	0
leader-3	263	33	1	1
leader-4	o.t.	o.t.	5	0
leader-5	o.t.	o.t.	11	1
leader-6	o.t.	o.t.	27	3
leader-7	o.t.	o.t.	55	4
leader-8	o.t.	o.t.	112	9
leader-9	o.t.	o.t.	232	17
leader-10	o.t.	o.t.	476	35
leader-11	o.t.	o.t.	936	69
leader-12	o.t.	o.t.	o.t.	244
leader-13	o.t.	o.t.	o.t.	668
leader-14	o.t.	o.t.	o.t.	o.t.

Figure 8.8: Run times in seconds using non-incremental and incremental hashing for the dining philosophers (top-left), the optical telegraph (top-right) and the leader election protocol (bottom).

8.5 Experiments on Multi Agent Planning

In the experiments we used the same system as for the experiments on state reconstruction. We applied our system to randomly generated MAMPs (cf. 5.5) with 5, 10, or 15 agents and 10, 20, 50, or 100 jobs. The number of resources was set to 100. Each job requires between 10 and 100 steps to be done and up to 10% of all resources. Our goal is to prove, that the combination of planning and simulation leads to better solutions than pure simulation. To do this, for each MAMP m, we first solve m ten times with pure simulation. Then we solve m ten times with the combination of planning and simulation. In both cases, we measure the number of parallel steps of the solution and the total time in seconds spent for planning. A planning phase may be at most 30 seconds long. If during planning a memory limit of 500 MB is exceeded or the open set becomes empty, the phase ends even if the time limit was not reached. A simulation phase executes at least 20 and at most 100 parallel steps. Additionally a simulation phase ends, if at least as many agents are idle, as were at the beginning.
We expect that in the average case the combination of planning and simulation gives better results (in terms of the number parallel steps), that justify the additional time spent on planning. The evaluation function that counts the number of taken jobs is expected to lead to a maximum of parallelism, since more taken jobs imply less idle threads. Table 8.9 shows the results. Each row represents ten runs on the same MAMP instance. The column *type* indicates the type of run, i.e. either pure simulation (*sim*) or combination of planning and simulation (*plan*). The columns *min*, *max* and μ show the minimal, maximal and average number of parallel steps needed for a solution of the respective MAMP. Finally, *pt* indicates the average time in seconds used for planning.

In almost all cases the average solution quality increases if planning is used. The only exception is the instance with 10 agents and 20 jobs, where the heuristic estimate seems to fail. Note that clashes may lead to worse solutions, since they cause an agent to waste one step realizing that the assigned job is already taken. For all other cases however, we have improvements between 6 and 59 parallel steps in the average. The total planning times range between 26 and 30 seconds, which seems odd, since each planning phase can be up to 30 seconds long. However, this can be explained by the fact, that only in the first planning phase all agents are idle. If in the subsequent planning phases only a small fraction of all agents is idle, the phases may get very short because the model checker can quickly decide, whether new jobs can be assigned or not. If for example we assume that each parallel step corresponds to one minute in reality, then the time spent for planning is negligible compared to the time gained through the improved solution quality.

type	agents	jobs	min	max	μ	pt
sim	5	10	262	315	286	0
plan	5	10	241	315	255	29
sim	5	20	517	616	546	0
plan	5	20	483	527	510	27
sim	5	50	1081	1233	1149	0
plan	5	50	1084	1187	1143	29
sim	5	100	2426	2573	2478	0
plan	5	100	2310	2501	2426	26
sim	10	10	259	266	261	0
plan	10	10	136	286	246	28
sim	10	20	389	465	432	0
plan	10	20	412	467	460	30
sim	10	50	754	902	839	0
plan	10	50	771	856	814	30
sim	10	100	1732	1871	1809	0
plan	10	100	1761	1858	1783	30
sim	15	10	260	263	261	0
plan	15	10	187	236	216	30
sim	15	20	385	463	417	0
plan	15	20	382	410	387	30
sim	15	50	675	896	767	0
plan	15	50	711	747	725	30
sim	15	100	1528	1728	1607	0
plan	15	100	1497	1600	1548	30

Table 8.9: Experimental Results in Multiagent Manufacturing Problems.

Chapter 9

Conclusion

In the following, we summarize the contribution of the work at hand and give an outlook on possible further work based on these results.

9.1 Contribution

The thesis addresses the verification of C++ programs. In contrast to other approaches to software model checking, our intention is to avoid the construction of formal models from program source code. Instead, we intend to model check C++ implementations directly.

Analysis of the Status Quo

The thesis gives a concise introduction to model checking. The elevator example from Chapter 2 illustrates the complexity of model checking even for small-scale systems. Moreover, the example illustrates interesting properties of the system, how they are formalized, checked and enforced in the model. In its succeeding course, the thesis identifies the verification of software implementations as a modern branch of model checking, while it points out the important differences to classical approaches that target the verification of system specifications rather than their implementation.

The need for software model checking is motivated by highlighting its advantages over traditional verification techniques such as testing. This claim is supported by real-life examples, such as the Mars Path Finder bug, which passed unnoticed through all phases of the development process.

Also, we introduce the most popular software model checkers and discuss the advantages and drawbacks of their approach. Our observation is that the majority of existing tools does not adequately handle the complex semantics of modern programming languages, such as Java or C++, and rely on an abstraction of actual source code instead. This seems contradictory to the idea of software model checking as a technique that aims to find errors that result from details of the software's implementation.

StEAM

As the works main contribution, the C++ model checker StEAM was created. By extending a virtual machine, the tool constitutes the first model checker for C++ programs, which does not rely on an abstract model of the program. The approach of using an underlying virtual machine for the exploration of a program's state space eliminates several problems of model based program verification. Firstly, by bypassing a modeling process, there is no danger that the user may introduce new errors in the model, which are not present in the actual program. Also, since we investigate real executables compiled from program source code, it is assured that the formal semantics of the program are correctly reflected. Thus, we know that each error found is also in the actual source code and that each error in the program can be found (assuming that a sufficient amount of time and memory is available). In its current version, the tool is already capable of verifying simple programs and of finding errors in larger ones. Note, that to truly verify a program, it's entire state space needs to be explored. In contrast to the majority of other software model checkers, StEAM is capable to find low-level errors such as illegal memory accesses, because it can evaluate the result of each machine instructions before they are executed.

StEAM is not the only program model checker that keeps the internal representation of the inspected program transparent to the user. Tools like Bandera and JPF1 (cf. sections 2.4.3,2.4.3) use a custom-designed formal semantic to translate source code either to some existing modeling language such as *Promela* (cf. Section 2.3.1). We have seen, that this approach yields two major drawbacks: on the one hand, this implies a considerable overhead for writing a translator from source code to the modeling language. These translators often do not even cover the full semantics of the respective programming language. On the other hand, the used modeling languages are often not appropriate to represent programs, as they were mainly designed for the verification of high-level protocols rather than actual software implementations.

Moreover, StEAM is not the only model checker which uses the actual machine code of the checked program. Tools such as CMC and VeriSoft (cf. sections 2.4.3 and 2.4.3) augment the actual program with a model checking environment and compile it to a model checking executable. This bypasses the need to devise a custom compiler or a machine-code interpreting infrastructure (like a virtual machine). However, this approach is not capable to catch low-level errors such as illegal memory accesses, since the program's machine code is executed on the physical hardware of the executing computer.

More similar to StEAM are the approaches of JPF2 and Estes (cf. sections 2.4.3). JPF2 uses its own Java virtual machine, for which it is probably easy to build a model checking algorithm on top. Yet, the design and implementation of the virtual machine implies a large overhead for the developers of the model checker. Also, such a machine is not thoroughly field tested for correct functioning - in contrast to, e.g., the official Java Virtual Machine, which is used by millions of people every day. Likewise, StEAM builds on the existing virtual machine IVM, which is known to reliably run complex applications such as Doom. Moreover, JPF2 is limited to the verification of Java programs, although most industrial applications - including those of the NASA - are written in C++.

The tool Estes uses an approach most similar to that of StEAM, as it builds the model checking algorithm on top of an existing and widely used tool, namely the GNU debugger GDB. Since GDB supports multiple processor levels, it is also capable to search for low-level errors that occur only in the machine code of a specific architecture. However, the currently available literature [MJ05] neither gives a detailed technical description on the architecture of Estes, nor does it conduct exhaustive experiments. This makes it, *yet*, hard to judge the possibilities of Estes and how feasible the approach is.

9.1.1 Challenges and Remedies

State Explosion

Verifying compiled source code yields two major challenges. The first is the *state explosion problem*, which also applies to classical model checking. This combinatorial problem refers to the fact, that the state space of a system grows exponentially with the number of involved components. Like other model checkers, StEAM needs methods to alleviate this problem. As the most promising approach, we have used heuristic search. The benefit from using heuristics in model checking is threefold: Firstly, since less states need to be memorized, the memory requirements are reduced. Second, because less states are visited, the error is found faster. Third, heuristic search yields shorter counter examples than, e.g., depth-first search. Shorter counter examples can be tracked down more easily by a user, which facilitates detecting the error in the program source code.

The use of heuristic search is not a novel idea. In fact StEAM inherits some of its heuristics from HSF-Spin (cf. Section 2.3.1) and JPF2. However, we also devised new heuristics, such as *lnb* which performs considerably better in finding deadlocks than the *most-blocked* heuristic (cf. Section 8.2). Moreover, we introduce heuristics such as *rw* and *aa*, which favor paths that maximize the access to global variables. We also propose the number of processes alive as a factor which can be combined with an arbitrary heuristic. The intuition is, that only alive threads can cause errors.

The experimental result in Section 8.2 are encouraging and are to a large extend in line with our expectations.

State Size

The second major challenge of verifying compiled code lies in the size of the state description. Since we need to memorize the full information about a program state, including dynamically allocated memory, the state description may grow to an arbitrary size. Firstly, this may aggravate the state explosion problem, since we have a higher per-state memory requirement. This problem can be alleviated by an incremental storing of states: since state transitions usually only change a small fraction of the state components, we only need to store these differences for the successor state. Also we devised state space search based on state reconstruction, which identifies states through their generating path.

The latter approach is - in similar form - also used in other works such as [MD05], which builds the search tree for colored petri nets by identifying intermediate nodes through the executed transition. However, for StEAM state reconstruction seems particularly appropriate, since the execution of the machine instructions along the path from the root to the reconstructed state is quite fast due to the IVM which was mainly optimized for speed. This results in a particularly worthwhile time- space-tradeoff. The experimental results in Section 8.3 support this.

Another problem of the large state description lies in the hashing effort for a state. Even, if a transition makes only marginal changes to a state, a conventional hash function needs to look at the entire state description to compute its hash code. This can make hashing the most expensive part of state generation. To overcome this problem, we devised an incremental hashing scheme based on the algorithm of Rabin and Karp.

To the best of the authors' knowledge, there is little research on reducing the hashing effort in model checking. The reason for this is presumably the fact than most other tools are based on abstraction, which yields relatively small state descriptions. Hashing is mentioned in [MD05] though. Here, the approach is to use a heap canonicalization algorithm which minimizes the number of existing heap objects that change their position in the canonical representation when new objects are inserted into the heap or existing ones are removed. This also minimizes the number of heap objects whose hash code has to be re-computed. The fundamental difference to our approach is, that when the content of a heap object changes, its hash code must always be fully computed, even if the change was marginal. In contrast, the incremental hashing scheme devised in Chapter 6 can also hash single objects incrementally.

9.1.2 Application

The functionality of StEAM was shown by model checking several simple C++ programs. These range from implementations of classical toy-models such as the dining philosophers to simple real-world applications like the control software for a cash dispenser. Note, that there is an additional example given in the appendix, which deals with the control software of a snack vendor. We are able to find deadlocks, as well as assertion- and privilege-violations. Applying heuristic search allows us to find errors with lower memory requirements and shorter error trails. The use of state-reconstruction considerably reduces the memory requirements per generated state, which allows us to explore larger state spaces with the same algorithm. The problem of large state descriptions is alleviated by incremental hashing, which can significantly speed up the exploration.

Additionally, we proposed a method to use StEAM as a planner for concurrent multi-agent systems, where simulation and exploration of a system is interleaved in a very natural way. The idea of applying model checking technology for planning is appears to be a logical consequence, since the two fields are quite related (cf. Section 5.1). The approach also takes a unique position in planning, since by planning on compiled code, we avoid the generation of a formal model of the system: the planning is done on the same program, which controls the actual agents.

The new model checker raised some attention in the model checking community. We have already received requests from *NASA - Ames Research Center* and *RTWH Aachen*, who would like to use StEAM on their own test cases.

9.1.3 Theroretical Contributions

We aim at giving a very general view on model checking as a verification method for state based systems. The conversion between the two graph formalisms in section 2.1.1 emphasizes, that model checking approaches rely on the same principles, independently of the underlying formalism or the field of application.

In chapter 6, we give a formal framework for incremental hashing in state space search.

Incremental hashing is an important issue in program model checking, but can also be applied to any problem domain that is based on state space exploration. As an important part of the framework, we devise efficient methods for incremental hashing on dynamic and structured state vectors.

Moreover, we make general observations about the asymptotical run times of hash computation in several domains and analyse time/space tradeoffs which invest memory to pre-calculate tables that allow an asymptotical improvement for the hash computation - e.g., from $O(n)$ to $O(1)$.

For state reconstruction, we give a general algorithm that is applicable to any kind of state space search. Whether it makes sense to apply the algorithm depends on two factors: the size of the state descriptions and the time needed to compute a state transition.

For the planning approach, we devised the multi-agent manufacturing problem (MAMP), as an extension of job-shop problems. Furthermore, we give a novel technique which interleaves planning and simulation phases based on the compiled code which controls the environment. The resulting procedure *interleave* (cf. Algorithm 5.4) is applicable to all domains, whose description allows exploration as well as simulation.

9.2 Future Work

StEAM continues to be a running project and the source code will be publicly available in the near future. We give an outlook for further development the tool may undergo in the future.

9.2.1 Heuristics

Heuristic search has shown to be one of the most effective methods to overcome the state explosion problem. With the appropriate heuristic, it is possible to quickly find an error even in programs that exhibit a very large state space. Hence, an important topic will be to devise new and improved heuristics.

Pattern Databases Of particular interest are pattern databases (PDBs). Here, the state space is abstracted to a degree that allows an exhaustive enumeration of the abstract states by an admissible search algorithm such as breadth-first search. The generated search tree is used to build a table which associates with each abstract state the distance to the abstract goal state (or error state). In a subsequent exploration of the original state space, the distance values from the table serve as heuristic estimates for the original states. PDBs were quite successfully applied in puzzle solving and planning. However, the application to program model checking imposes some challenges:

First, one must find an appropriate abstraction. An easy solution would be to select a subset of the state components. However, a state of StEAM is merely given by the stacks, CPU-registers, data-/bss-sections and the lock-/memory-pool, which allows to define only $2^6 = 64$ distinct abstractions. A more informative heuristic can be obtained by abstractions at the source code level, e.g. by restricting the domain of variables. The implementation would be technically challenging, since during exploration the abstractions must be mapped from the source code level to the corresponding memory cells.

Second, in contrast to AI puzzles, not all program transitions are invertible. For instance, we cannot define the converse of a program instruction, that writes a value to a variable x, since the previous value of x is not a-priori known. The classical applications of pattern databases [CS96, CS94] rely on a backwards search in the abstract state space, starting at the abstract goal state. In domains, where state transitions are not invertible, the search in the abstract state space must maintain a list of all predecessors with each generated state. Since duplicate states - and hence multiple predecessor states are common in concurrent programs, this may considerably increase the memory requirements for the abstract state space enumeration.

Trail-Based Heuristics

Prior to the existence of StEAM, the author did some research on trail-based heuristics. These exploit a complete error state description from a previous inadmissible search to devise consistent heuristics. These heuristics can then be used in combination with an admissible heuristic search algorithm such as A* or IDA* to obtain a shortest error trail.

One example of a trail-based heuristics is the *hamming-distance*, which counts the number of unequal state components in the current and the error state. Another example is the FSM-distance, which abstracts states to the program counters of the running threads. Hence, the FSM-distance is related to pattern databases. The latter however, generally require a complete enumeration of the abstract state space to compute the table of heuristic values, while for the FSM-distance this information can be extracted from the compiled code of the program through static analysis.

The author implemented both heuristics for the Java program model checker JPF2. Some experimental results have been published [EM03]. We passed on an implementation for StEAM, as we wanted to concentrate on fundamentally new concepts in our research. Anyhow, the generation of short counter examples remains an important aspect in model checking, hence implementation of the discussed trail-based heuristics may be an option for future enhancements of the model checker.

9.2.2 Search Algorithms

StEAM already supports the uninformed search algorithms depth-first and breadth-first search, as well as the heuristic search algorithms best-first and A*. The memory-requirement of the state exploration remains an important topic in program model checking. Hence, StEAM should offer additional algorithms that have a stronger bias on memory efficiency.

Iterative Deepening

Iterative deepening algorithms such as iterative depth-first (IDFS) or IDA* (cf. Chapter 3) are particularly attractive for model checking, as they are both optimal and memory-efficient. As an advantage iterated deepening algorithms only need to maintain a small set of open states along the currently explored path. However, to be truly effective, the algorithms must be integrated with hash compaction methods which provide a compact representation of the closed states.

StEAM already provides a rudimentary implementation of IDFS. Currently each state must be fully expanded, instead of generating just one successor state. For an outgoing degree of k, this implies that at a search depth of n, the size of the open list is $n \cdot k$ instead of n. The problem is, that value k for a state cannot be determined before the state is fully expanded, because k is a function of both the number of running threads and the non-deterministic statements.

External Search

When a model checking run fails, this is usually because the space needed to store the list of open states exceeds the amount of available main memory (RAM). The amount of storage on external memory devices such as hard disk is usually a multiple of the machine's main memory. At present, the standard personal computer is equipped with 512MB RAM and a 80GB hard disk, i.e. the secondary memory is 160 times the amount of available RAM. An obvious approach is to make that memory usable for the state exploration.

Implicitly, this is already possible through the virtual memory architecture of modern operating systems. Unfortunately, the virtual memory management has no information about the locality of the states, we store - i.e. it will be common that each consecutive access to a stored state causes a page fault and thus an access to the secondary memory. Obviously this would slow down the exploration to a unacceptable degree. For an efficient use of external memory, the data exchange between primary and secondary memory must be managed by the search algorithm. One of the first approaches to *external search* applies BFS with delayed duplicate detection and uses secondary memory to efficiently solve large instances of the Towers of Hanoi puzzle [Kor03] and the $n^2 - 1$-puzzle [KS05]. In a different line of research, an external version of A* is used to solve the 15-puzzle [EJS04]. The external A* was later integrated into SPIN [JE05], yielding the first model checker capable of using external search.

Since memory limitations impose an important challenge in software model checking, integrating external search algorithms into StEAM would be very interesting.

9.3 Hash Compaction and Bitstate-Hashing

Compacting hash functions allow a memory efficient storing of the set of closed states. Here, we essentially distinguish between hash compaction and bitstate-hashing.

In hash compaction, a second hash function is used to compute a fingerprint of the state s, which is then stored in a hash table. A state is assumed to be a duplicate of s, if its fingerprint is already stored at the corresponding position in the hash table.

In bitstate-hashing the state is mapped to one or more positions in a bit-vector. A state is assumed to be a duplicate, if all corresponding bits in vector are already set.

With compacting hash functions, more states can be visited, although completeness of the search is sacrificed: If two distinct states are falsely assumed to be duplicates, the search tree is wrongly pruned and an error state may be missed. However, statistical analyses have shown that the probability for this is very low [Hol96].

The use of compacting hash functions is of particular interest in combination with iterated deepening search (cf 9.2.2), because here the list of open states is generally small compared to the number of closed states.

StEAM already supports a version of bitstate-hashing, which maps the state to a single position in a bit vector using a single hard-coded hash function. In the future, the number of bit positions should be parameterizable, because mapping the state to more than one position will significantly reduce the risk of hash collisions and thus the danger of missing an error state. Using an arbitrary number of bit positions per state also implies that the corresponding hash functions must be automatically generated.

9.4 State Memorization

To reduce the memory requirements of the exploration, states should be stored in an incremental fashion. In its current form, StEAM already uses backwards pointers to components of immediate predecessor states. This can alleviate the redundant storing of states, but it cannot fully prevent it: First, two states may share identical components, although one is not an immediate predecessor of the other. Second, components get fully stored, even if they differ only slightly from the corresponding component of the predecessor state. This leads to particularly strong redundancies, e.g., if the respective component constitutes a large array in which the state transitions change only one memory cell at a time. In the future one may consider a state storage, which implies less redundancies. It is important however, that the reduced memory requirements do not come at the cost of a significant slowdown of the exploration.

Collapse-Compression The model checker SPIN uses a sophisticated approach called collapse compression [Hol97c] to store states in an efficient way. Here, different state components are stored in separate hash tables. Each entry in one of the tables is given a unique index. A whole system state is identified by a vector of indices that indicate the corresponding components in the hash tables.

The use of collapse compression prevents the repeated storing of a component. However, two fully stored components may still expose minimal differences.

Memorizing Changes

Alternatively, states can be represented by the differences to their predecessor state. To some degree this is already implemented by the state reconstruction of StEAM (cf. Section 7), states are identified through their generating path. However executing the respective machine instructions is most probably slower than applying the resulting changes directly to the state.

As another disadvantage, not all machine instructions are invertible, because the converse of writing a new value to a memory cell c depends on c's old content. This makes it impossible to directly backtrack from a state s to its predecessor p. Alternatively, for each memory cell c whose value was changed by the transition from p to s, we can memorize the difference $\delta_c(s,p) = c[s] - c[p]$ of the old and new value. This allows us to backtrack from s to by calculating $c[p] = c[s] + \delta_c(s,p)$ for each changed memory cell c.

9.4.1 Graphical User Interface

In the long term, research in model checking must yield useful tools, which are also accessible for practitioners. StEAM already provides a certain degree of user-friendliness, since its use does not require knowledge about specialized modeling languages such as Promela. Moreover, the counter examples returned by StEAM directly reflect the corresponding execution of the investigated program, rather than that of a formal model. The latter would make it much harder to track down the error in the actual source code.

To obtain a broad acceptance in the industry, it will be essential to devise an integrated graphical user interface which allows an easy and intuitive use of the tool.

9.5 Final Words

Beyond the comparisons of the various approaches, the work at hand hopefully gave a clear impression of how model checking may help to ease the software development process. In particular, it is desirable that the industry becomes aware of the potential that model checking-assisted software development offers as an alternative to the state-of-the-art engineering process, as in the latter, the software engineer invests a considerable amount of time on formal specifications and an even more time on browsing code in the attempt to detect the cause of an error. Software model checking can provide the engineer with detailed information about the cause of an error, which makes it much easier to track down and correct the corresponding piece of code.

The approach of extending an existing virtual machine, not only ensures a correct interpretation of the semantics of a program - also the internals of the model checking process remain transparent to the user, who in turn is more likely to accept the new technology. We hope that in the future more researchers become aware of this fact.

Appendix A

Technical Information for Developers

In the following, we give some detailed information about the implementation of StEAM. It is meant to make it easier for developers to get known to the source package of the model checker.

A.1 IVM Source Files

The original package of *ivm* includes the virtual machine and a modified version of the gcc compiler *igcc*, which produces the ELF executables to be interpreted by *ivm*. As stated before, the approach of StEAM does not require changes to igcc, so only the source files for the virtual machine are of relevance. Moreover, only the files located in the sub folder *ivm*/*vm* need to be regarded. These include:

icvmvm.c

In the original package, this file contained the loop which reads and executes the program's opcodes. For StEAM, this was replaced by a loop, which expands program states according to the chosen algorithm. The original program execution is only done until the main-thread registers to the model checker. At this point, the contents of the stack, the machine, the global variables and the dynamic memory are memorized to build the initial state (i.e. the root state in the search tree).

The source file also holds the functions *getNextState*, *expandState* and *iterateThread*. The first function chooses the next open state to be expanded according to the used search algorithm. The second function expands a given state and adds the generated successors to the open list. The third function iterates a given thread one atomic step by executing the corresponding machine instructions.

icvmsup.c

This file implements the instructions of the virtual machine in a huge collection of mini-function. Due to its size of more than 140000 lines, it is compiled to 14 separate object files (icvmsup.o, icvmsup1.o,...,icvmsup13.o) before linking. Compiling icvmsup takes quite long (several minutes on a 1.8GHz machine). This is quite inconvenient in cases, where a *make clean* needs to be done (e.g. when a constants in a header file were changed), as the original makefile will also delete the icvmsup object files, even if the machine instructions are not affected by the change. In this case it may be convenient to move the object icvmsup object file to a temporary folder before cleaning up and moving them back before the subsequent *make all*. The source file icvmsup.c was not modified during the development of StEAM. However, the file is useful to find out about the meaning of a certain instruction. In icvmsup.c, an instruction can be located through its corresponding opcode. For example, if we disassemble the dining philosophers from Chapter 8 with *iobjdump -d philosophers | less*, we find a fraction of machine code:

```
                  .
                  .
                  .

20:     0200 7c2c 0000  bsrl 2c9c <___do_global_ctors>
26:     a8eb 0800       movel (8,%r2),-(%sp)
2a:     a8eb 0400       movel (4,%r2),-(%sp)

                  .
                  .
                  .
```

We may want to find out about the instruction *brsl <x>* (branch sub routine long). Note that iobjdump displays opcodes according to the hi/low order of the underlying machine. In our case, we have an Intel-processor with Little-Endian notation, which means that the corresponding opcode is 0x2 rather than 0x200. In icvsump.c we find a line starting with:

```
_If1(_sCz2,2,s32) ...
```

The instructions for *ivm* are generated through a macro, whose first parameter is the opcode with a leading "_sCz". Note that the opcode is appended without any leading zeros. We can easily interpret the implementation of the instruction, even without knowing how the macro translates to compilable c-code:

```
{{(R_SP-=8,WDs32(R_SP,R_PC+6,0));R_PC=(is32(2)+R_PC);}
```

First, the stack-pointer (R_PC) is decremented to reserve space for the return-address of the called sub routine. Then, that address is stored at the new position of stack-pointer through the macro *WDs32* (write signed 32-bit data). The return address corresponds to the current program counter plus 6 - the length of the bsrl instruction. Afterwards, the program counter is set to the the current position plus a 32-bit offset that follows the opcode. Apparently, the macro *is32(y)* reads a 32-bit parameter from address of the program counter plus y.

cvm.h

This header defines the basic data types needed for the virtual machine, such as the machine registers. The macros used by the mini-functions in icvmsup.c - such as WDs32 - are also defined here. As a big convenience, all memory read- and write-accesses are encapsulated through the macros of this header. This enabled us to record all such accesses by simply enhancing the macros. On the one hand, this allow us to determine changes made by a state transition without fully comparing the pre- and successor state. On the other hand, we can capture illegal memory accesses before they actually happen.

A.2 Additional Source files

The following source files were added for StEAM.

include/*

The name is somewhat misleading, as this folder also contains ".c" files. More precisely, it contains the AVL-Tree package used for the lock- and memory pool of states in StEAM. Using the AVL trees may not have been the best choice for storing the pools, as their handling is tedious and a simple hash table may be faster in many cases. Developers are hereby encouraged to replace the AVL trees with a more suitable data structure.

mysrc/*

This folder holds the source files exclusively written for StEAM - these include:

IVMThread.h | .cc

This is the class definition for threads in StEAM. All new thread classes must be derived from IVMThread. A thread registers to the model checker by executing a TREG command-pattern in its start-method. A second TREG statement tells StEAM, it has reached the *run()*-method of the thread.

mfgen.c

This is the source code of the tool, which is parameterized with the names of all involved source files of the checked program. In turn generates a makefile for building the *ivm*-executable that can be model checked by StEAM.

extcmdlines.cpp

This is the source code annotation tool. Before the investigated program is compiled, its source is annotated with line number information and information about the name of the source file through command patterns. The annotation is automatically done in the makefile generated by mfgen and , hence, remains transparent to the user. It must be assured though, that extcmdlines.cpp is compiled to an executable called *extcml* must lie in the system path.

The source annotation tool is actually a workaround, as StEAM does currently not use the debug information that can be compiled into ELF files (e.g. using the -g option on gcc). Writing a parser for this information would have cost too much time considering the manpower available for the project.

Developers are encouraged to add this functionality to StEAM, as it would make the annotation redundant and speed up the entire model checking process.

pool.h | .c

These files define the functions needed for the lock- and memory-pool of StEAM's state description.

icvm_verify.h | .c

These source files are the core of StEAM as they define the command patterns, data structures and functions needed by the model checker. A command pattern is generated by the macro INCDECPATTERN, which translates into a senseless sequence of increment and decrement instructions. The parameters of a pattern are realized by assigning their value to a local variable. After parsing the command pattern, StEAM can obtain the values of the parameters directly from the stack.

Most functions defined in icvm_verify.c relate to program states. This includes, cloning of a state, comparison of two states, deletion of states etc.

Moreover, icvm_verify.c implements the functions of StEAM's internal memory management. The management is encapsulated through the functions *gmalloc(int s)* and *gpmalloc(int s, char * p)*, which allocate memory of a given size or of a given size plus a given purpose. When StEAM is invoked with the parameter *-memguard*, all memory regions allocated with gmalloc or gpmalloc are stored in a private memory pool. NOTE: This refers to the memory pool of the model checker and must not be confused with that of the investigated program. It is possible to print the contents of the memory pool using the function *dumpMemoryPool()*. StEAM's internal memory management is useful to e.g. detect memory leaks in the model checker. For instance, this was important during the implementation of state-reconstruction (cf Chapter 7). Here, states are frequently deleted after being replaced by a corresponding mini-state. If a few bytes of the state are not freed, the resulting leaks will quickly exceed the available memory.

The source file icvm_verify.c also implements the currently supported heuristics through the function *getHeuristicEstimate*. The latter function is called by *getNextState()* in icvmvm.c, which chooses the state to be expanded next according to its heuristic value.

Note that for simplicity undirected search is also implemented through heuristic values. For BFS, the heuristic estimate of a state corresponds to its depth in the search tree. For DFS the estimate is determined by subtracting the depth from a constant which is greater than any search depth we may ever encounter.

steam.c

This is the frontend of the model checker, which calls the executable of the modified virtual machine. Environment variables are used to pass the parameters.

hash.h | .c

These files implement the (non-)incremental hashing in StEAM. The hash function is parameterized with a bitmask, which determines the hashed components of the state description. The frontend of steam currently only gives two choices: Either only the cpu-registers are hashed (default) or the entire state (-fullhash). The cpu registers were chosen for the partial hash option, since they constitute the state component that is most likely to be changed by a state transition. Developers are encouraged to try other subsets of components.

A.3 Further Notes

The functions *getNextState*, *expandState*, *iterateThread* and *printTrail* actually do not belong in the file icvmvm.c and it would be a good idea to move them to icvm_verify.c.

When StEAM is started with the *-scrtrl* option, the error trail is printed as the actual source lines of the checked program (rather than just the line numbers). For this, StEAM makes use of the Unix-commands *cat* and *grep*. Hence, the option will not work on systems where these commands are not available. It would be desirable to rewrite the *printTrail* function such that it no longer requires any external programs.

As a definite drawback, states must currently be fully expanded. The problem is, that it is not a-priori known, if during thread iteration a non-deterministic statement is encountered. In the latter case, execution of a thread yields more than one successor. For this reason state reconstruction is currently not available for programs involving non-deterministic statements. Also the full expansion is disadvantageous for depth-first variation, such as DFS and IDA*, since at depth d the model checker needs to memorize $d \cdot o$ states instead of d - where o is the outgoing degree of a node in the search tree. Hence, a big improvement would be to add the possibility to directly generate the i-th successor of a state.

Appendix B

StEAM User Manual

B.1 Introduction

StEAM, (State Exploring Assembly Model checker) is a model checker for native concurrent C++ programs. It extends a virtual machine - called ICVM - to perform model checking directly on the assembly level. This document gives you the basic information needed to install and use *StEAM*. The example programs included in the installation package are supposed to show the current capabilities of *StEAM*. However, we encourage you to try to verify your own programs. This manual will also explain the steps needed to prepare a piece of software for verification with *StEAM*. For detailed information about the internals of the model checker, we refer to [ML03].

B.2 Installation

The installation instructions assume that StEAM will be installed on a Linux system using gcc. *StEAM* is based on the "Internet Virtual Machine" (IVM). The IVM package comes with source code for the compiler and the virtual machine (vm) itself. In the course of the *StEAM* project only the vm needed to be enhanced, while the compiler was left unchanged. The task of building *igcc* - the compiler of IVM, will hence be assigned to the user. The rest of the model checker, including the modified virtual machine can either be obtained as a binary package for Linux or as a source package. The source package can be compiled for use on Linux and Cygwin (a Linux-like environment for Windows).

B.2.1 Installing the Compiler

The original source package of IVM can be downloaded from the project's home page [1]. The package was found to compile with older versions of gcc (e.g. 2.95.4). However, later versions such as 3.4.4 require considerable changes to the makefile in order to build igcc. You may obtain an adapted version of the source package, which allows to build the

[1]http://ivm.sourceforge.net/

compiler on more recent versions of gcc from the StEAM-page[2]. Note that this applies only to igcc. If you want to compile the original virtual machine on modern versions of gcc, additional changes to the source package may be necessary.

Once you have build igcc, make sure that the directory containing the binaries (i.e. igcc, ig++, ...) are included in the PATH variable.

B.2.2 Installation of *StEAM*

In the following, we assume that you have downloaded the binary installation package of StEAM. If you prefer to compile the source package, please refer to the installation instructions given there. The binary installation package of *StEAM* contains the enhanced virtual machine, the frontend of the model checker, the source annotation tool, the makefile generator and some header files. Unpack the contents of the package to an arbitrary directory e.g. "/home/foo/".

Next, set the following environment variables:

```
PATH=$PATH:/home/foo/steam_release/bin

STEAM_INCLUDE_PATH=/home/foo/steam_release/include

C_INCLUDE_PATH=$STEAM_INCLUDE_PATH

CPULS_INCLUDE_PATH=$C_INCLUDE_PATH
```

B.3 A First Example

If you followed the instructions in section B.2, you should now be able to check one of the example programs. Go to the directory:
/home/foo/steam_release/models/philo. Here lies a C++-implementation of the dining philosophers problem. If you list the directory, you will notice the following files:

```
Makefile
Philosopher.cc
Philosopher.h
philosophers.c
```

Besides the Makefile, there are the files *Philosopher(|.h|.cc)* which describe the c++-class of a philosopher-thread. If you look at the content of Philosopher.cc (e.g. with *less Philospher.cc*), you will notice, that it basically does locks and unlocks two global

[2]http://ls5-www.cs.uni-dortmund.de/∼mehler/uni-stuff/steam.html

variables which were passed to the constructor function. The file *philosophers.c* contains the main() method which generates and starts the threads.

Compile the program by typing:

```
make
```

This will first apply annotations to the *.c* and *.cc* source files. After this, the example program is compiled and linked with *ig++* - the compiler of IVM. Do another directory listing and you will see the following:

```
Makefile
Philosopher.cc
Philosopher.cc_ext.c
Philosopher.h
Philosopher_ex.o
philo
philo_ex.o
philosophers.c
philosophers.c_ext.c
```

The source files containing an “ext|.c|.cc” in their name were produced by the source annotation tool. The file *philo* is the IVM-object file compiled from the annotated source files. Start the model checker on the example program by typing:

```
steam philo 2
```

This will start the model checker with the default parameters on the compiled program *philo* parameterized with ’2’. In the case of the dining philosophers, this means that 2 instances of the thread class Philosopher are created. The typed command will produce the output of some standard information from the model checker, followed by the report of a detected deadlock, an error trail and some statistics - such as the number of generated states, memory requirements etc. Each line of the error trail tells us the executed thread, the source file and line number of the executed instruction. Here, Thread 1 refers to the main program, while Thread 2 and Thread 3 correspond to the first and second instance of Philosopher. However, the error trail may seem a bit lengthy for such a small program. Now type:

```
steam -BFS philo 2
```

This tells the model checker to use breadth-first (BFS) search (instead of depth-first which is the default search algorithm). The error trail produced by BFS will be significantly shorter than the first one. You may want to compare the source files with the information given in the trail to track down the deadlock. Alternatively, you can add the parameter *-srctrl* to the command line. This prints the error trail as actual source lines - rather than just as line numbers. Apparently, a deadlock occurs if both Philosophers lock their first variable and wait for the respective other variable to be released.

B.4 Options

StEAM is invoked by a call:

steam <[-optargi]> <progname> [<prog. arguments>], where

- <progname> is the name of the program to be checked.
- <prog. arguments> are the parameters passed to the program to be checked
- <[-optargi]> are optional model checker parameters.

```
steam -h
```

Prints out the list of available parameters. The parameters come from one of the following categories:

B.4.1 Algorithms

In the current version, *StEAM* supports depth-first search (default), breadth-first search (-BFS), depth-first iterative deepening (-IDFS) and the heuristic search algorithms bestfist (-BESTFIRST) and A* (-ASTAR). The latter two additionally require that a heuristic is given (see B.4.2).

Also note, that IDFS is only effective in combination with bitstate-hashing (see below).

B.4.2 Heuristics

For detailed information about each heuristic, we refer to [LME04]. The heuristic to use is specified by the parameter *-heuristic <hname>*, where *<hname>* is from one of the follwing groups.

Deadlock heuristics

These heuristics are specifically tailored to accelerate the search for deadlocks, but will in general not be helpful for finding other errors, such as assertion violations.
StEAM currently supports the deadlock heuristics: *mb, lnb, pl1, pl2, pb1* and *pb2*. For example:

```
steam -BESTFIRST -heuristic lnb philo 30
```

Uses best-first search and the *lock and block* heuristic to model check the dining philosophers with 30 instances of the Philosopher class. The BESTFIRST parameter may have been omitted in this case, because *StEAM* uses best-first as the default algorithm when a heuristic is given. You may compare the results of the directed search with these of undirected search (type 'steam philo 30' or 'steam -BFS philo 30').

Structural Heuristics

Rather than searching for a specific kind of error, structural heuristics [GV02] exploit properties of concurrent systems and the underlying programming language, such as the degree of thread interleaving and the number of global variable accesses. The currently supported structural heuristics are *aa,int* and *rw*.

B.4.3 Memory Reduction

State Reconstruction (experimental!)

When the option *-rec* is given, *StEAM* does not store generated states explicitly, but identifies each state by its generating path. This trades computation time for a reduced memory consumption. As to version 0.1 of StEAM, this option will not properly work with programs that make use of the non-deterministic range-statement (cf. Section B.6).

Lock-Global-Compaction

The parameter *-lgc* activates lock-global compaction, which allows thread-switches only after either a global variable was changed or a resource was locked. In experiments, this reduction method has shown to considerably reduce the memory requirements of a model checking run. However, it is yet not formally proven that lock-global-compaction preserves completeness.

B.4.4 Hashing

By default, StEAM only hashes the cpu-registers of a program state. Other options are:

Full Hashing

The option *-fullhash* instructs StEAM to hash the entire program state.

Incremental Hashing

Given the option *-inch*, the entire state description is hashed in an incremental fashion, which is usually faster than -fullhash alone. Note, that -inch implies full hashing.

Bitstate Hashing

When option *-bitstate* is given, closed states are not completely stored but mapped to a position in a bit-vector. Bitstate hashing should always be used with full hashing, as this minimizes the chance that StEAM will miss states.

B.4.5 Debug Options

These options are only interesting for developers that want to write their own version of StEAM.

GDB

The option *-GDB* starts StEAM in the GNU debugger gdb.

Memory Management

Passing then option *-memguard* activates StEAMs internal memory management. This is helpful to e.g. find memory leaks in the model checker (but not in the checked program).

B.4.6 Illegal Memory Accesses

Given the option *-illegal*, StEAM will explicitly look for illegal memory accesses (segmentation faults). As this may significantly slow down the exploration, it is not a default option.

B.4.7 Other Options

Simulation

The option *-sim* tells *StEAM* to simulate (execute) the program instead exploring its state space.

Time and Memory limit

You can set the maximum amount of memory in megabytes (*-mem <mb>*) and the maximum runtime in seconds (*-time <s>*) required by the model checking run. When one these limits is exceeded, the model checker prints its final statistics and terminates.

Log File

If you pass the option *-FSAVE* to the model checker, the output will be written to a file instead of the standard output.

Source Trail

When option *-srctrl* is given, the error trail is printed as actual source lines, rather than as line numbers. StEAM currently uses external Unix-commands to print single lines from source files. As a drawback, the printing of the trail cannot be interrupted using *ctrl-c*, as this terminates the external program rather than the model checker. This problem should be fixed in future versions.

No Trail

The option *-nt* suppresses the printing of the error trail - this is useful while doing experiments where the actual trail is of no interest.

B.5 Verfying your own Programs

In this section, we give a small tutorial, how your own code can be verified with *StEAM*.

B.5.1 Motivation

The desire to verify software may in principle arise from two reasons. Either we already have some faulty implementation of a system and want to use model checking to detect the error; or we are just starting with our implementation and want to use model checking to avoid the generation of errors while our software is being developed. In this tutorial, we want to concentrate on the latter case. Using assembly model checking in the implementation phase yields several advantages. On the one hand, it may reduce or even replace the efforts spent on verifying properties of a formal description of the program. Skipping the formal modeling will not only save a considerable amount of development time, also there is no need for the developer to learn the underlying description language (e.g. Promela). Furthermore, assembly model checking is able to find implementation errors which were not present in the software's specification.
The concurrency in *StEAM* can also be used twofold. Either, the investigated software itself describes a concurrent system. In this case, we define the processes in the software in terms of threads in *StEAM*. Or, the software itself is not concurrent. In the latter case, we can use threads to simulate the environment of our system.

B.5.2 The Snack Vendor

As an example, lets us assume we want to develop the control software of snack vendor machine. As its basic functionalities, the machine should accept that coins are inserted to raise the credit and buttons are pressed to buy a certain snack. Figure B.1 shows the bare bones of the software formed by two minimal functions - insertCoin() and pressButton().

When a coin is being inserted, the credit is raised by the appropriate value. When a button is pressed, the machine first checks if there is sufficient credit to buy the snack.

If this is the case, the price of the snack is subtracted from the credit. Two counters - *got* and *sold* - keep track of how much money was received and the total value of the snacks sold. A first property we would like to check is that the total value of sold snacks is always less or equal to the amount of money received, i.e. we want to check for the invariant: *(got>=sold)*. Currently, *StEAM* supports the detection of deadlocks and local assertions. However, we are also able to check for invariants by introducing an observer thread. The source code of this observer is shown in Figure B.2. As can be seen, a thread class in *StEAM* must be derived from the abstract class "IVMThread" and must implement the methods *start() run()* and *die()*. The start()-method makes preparations for the generated instance and must finally call the run()-method. The call to the run() method tells *StEAM* to insert the new thread into the running system. The run()-method itself must contain the actual loop of the thread. The die()-method is reserved for future versions of *StEAM* and currently has no effect - it must be implemented, but currently the body can be left empty. The statement *VASSERT(bool e)* is used to define local assertions. If the program reaches the control point of the statement and the expression *e* evaluates to 'false', *StEAM* returns the path that leads to this state and reports an error.

Note that we do not have to put our assertion into an infinite loop. We can rely on searching only for those error states, where the observer thread is first executed when the assertion does not hold. Thus, we avoid the generation of equivalent paths to the same error state.

Furthermore, we need to simulate the environment of our system - in this case, the users of the machine. One way to achieve this, is to introduce two derived thread classes - one that inserts coins and one that presses buttons. The header and source files for these are shown in Figure B.3 and B.4. Both classes use the statement *RANGE(varname n, int min, int max*, which causes a value between *min* and *max* to be non-deterministiclly chosen as the value of the integer variable *n*.

Although, in its current form, the source code can already be compiled to an IVM-executable, *StEAM* will not be able interpret the program correctly. First, the *.c* and *.cc* files need to be annotated with additional information. Fortunately, this remains mostly transparent to the user. The *StEAM* package comes with the tool *mfgen*. The latter takes as its parameter a set of source files and generates a Makefile which can be used to build the model-checkable file. In the directory where the sources for our vendor machine are located, simply type:

```
mfgen *
```

followed by

```
make
```

This will create the binary file "*vendor*".

B.5.3 Finding, interpreting and fixing errors

Now, start the model checker by typing:

```
steam -BFS vendor
```

After a (hopefully short) time, the model checker will print the error trail and report an assertion violation. Note that we have used breadth-first search (BFS) to detect the error but the default algorithm is depth-first (DFS). In most cases, DFS find errors faster than BFS, but the BFS is guaranteed to return the shortest trail. However, in some cases BFS also finds an error faster than DFS - this can happen, if the search tree is deep but the error is located at a shallow level. You may try to run the model checker again without the -BFS option, which will probably exceed the memory limit of your machine. Take some time for interpreting the error trail before you read on ...
It should be obvious, what happened: The button was pressed immediately after the credit was increased, but before the coin value was added to *got*. We can fix the error by declaring the two variable increases as an atomic block, i.e.:

```
    BEGINATOMIC;
    credit+=amount;
    got+=amount;
    ENDATOMIC;
```

When you recompile the program (*make*) - and start the checker anew, the error will no longer occur.

B.6 Special-Purpose Statements

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[3]. In Section B.5 we already used some of these statements for verifying the vendor machine. Now, we summarize statements, that are currently supported by *StEAM*.

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

[3]A following semicolon is optional.

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 $max - min + 1$ 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 terminates.

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.

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.

B.7 Status Quo and Future of *StEAM*

The model checker *StEAM* is an attempt to realize software model checking without abstraction and the need to manually construct models in specialized languages, such as Promela. In the current form, the tool is already able to find errors in small programs. The large search space induced by real programs remains the main challenge of software model checking, which may inhibit the finding of an error. Therefore, one of the main issues of future development lies in finding good heuristics. As shown in previous work [LME04], an appropriate heuristic is able to find errors fast, even if the underlying search space is huge.
Also we will improve the memory-efficiency of *StEAM* by optimizing its state representation. Future versions of *StEAM* will also support memory-efficient techniques such as bitstate hashing. In the long term our vision is to create a user-friendly application, which facilitates software- development and maintainance without the need for in-depth knowledge about model checking technology.

```
/***********************************************
 * vendor.c
 *
 * The bare bones of a snack vendor machine
 *
 * written by Tilman Mehler
 ***********************************************/

#include "vendor.h"
#include "icvm_verify.h"
#include "Inserter.h"
#include "Presser.h"
#include "Checker.h"

int credit=0;
int coin_values[3] = {10, 50, 100};
int price[3] = {50, 100, 200};
int sold=0;
int got=0;

void main() {

  BEGINATOMIC;
  (new Inserter())->start();
  (new Presser())->start();
  (new Checker())->start();
  ENDATOMIC;
}
void insertCoin(int amount) {
  if(credit+amount<=MAX_CREDIT) {
    credit+=amount;
    got+=amount;
  }
}
void pressButton(int n) {
  if(price[n]<=credit) {
    credit-=price[n];
    sold+=price[n];
  }
}
```

Figure B.1: Source code of the snack vendor software

```
/*************************************
 *
 * Checker.h
 *
 * Declarations for the Checker-class
 *************************************/

#ifndef CHECKER_H
#define CHECKER_H
#include "IVMThread.h"

class Checker:public IVMThread
{
 public:
  Checker();
  virtual void start();
  virtual void run();
  virtual void die();

};
#endif

/*************************************
 *
 * Checker.cc
 *
 * Definitions fot the Checker-class,
 * which makes an invariant-check
 *************************************/

#include "Checker.h"
#include "icvm_verify.h"

extern int credit;
extern int got;
extern int sold;

Checker::Checker() {}

void Checker::start() {
  run();
}
void Checker::run() {
  VASSERT(got>=sold);
}
void Checker::die() {}
```

Figure B.2: Source code of our observer thread

```
/*************************************
 * Inserter.h
 *
 * Declaratiins fot the coin-inserter
 * Thread-class
 *************************************/

#ifndef INSERTER_H
#define INSERTER_H
#include "IVMThread.h"

class Inserter:public IVMThread
{
 public:
  Inserter();
  virtual void start();
  virtual void run();
  virtual void die();
};
#endif

/*************************************
 * Inserter.cc
 *
 * Definitions fot the coin-inserter
 * Thread-class
 *************************************/

#include "vendor.h"
#include "Inserter.h"
#include "icvm_verify.h"

extern int coin_values[3];

Inserter::Inserter() {}

void Inserter::start() {
  run();
}
void Inserter::run() {
  int i=0;
  while(1) {
    RANGE(i, 0, 3);
    insertCoin(coin_values[i]);
  }
}

void Inserter::die() {}
```

Figure B.3: Source code of our observer thread

```
/**************************************
 * Presser.h
 *
 * Declaratins fot the button-presser
 * Thread-class
 **************************************/

#ifndef PRESSER_H
#define PRESSER_H
#include "IVMThread.h"

class Presser:public IVMThread
{
 public:
  Presser();
  virtual void start();
  virtual void run();
  virtual void die();
};
#endif

/**************************************
 * Presser.cc
 *
 * Definitions for the coin-inserter
 * Thread-class
 **************************************/

#include "vendor.h"
#include "Presser.h"
#include "icvm_verify.h"

Presser::Presser() {}

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

void Presser::run() {
  int i=0;
  while(1) {
    RANGE(i, 0, 3);
    pressButton(i);
  }
}
void Presser::die() {}
```

Figure B.4: Source code of our observer thread

Appendix C

Paper Contributions

In the following, I specify my contributions to the papers published in the course of my research.

Bytecode Distance Heuristics and Trail Direction for Model Checking Java Programs [EM03]. This paper is based on my master's thesis which was supervised by Stefan Edelkamp.

Introduction to StEAM - An Assembly-Level Software Model Checker [ML03] This is the first paper that gives a technical description of the software model checker *StEAM*. It was my idea to enhance the virtual machine *IVM* to a model checker. Moreover, I had devised and implemented:

- The enhancement of the virtual machine with multi-threading.
- The state description.
- The incremental storing of states.
- The two supported search algorithms (depth-first and breadth-first).
- The testing for deadlocks and assertion violations.
- The concept of "Command Patterns".
- The hashing.
- All models for the experiments.

Peter Leven helped in the initial installation of the IVM-package. Moreover, he implemented the lock- and memory-pool based on an existing AVL-tree package.

I also conducted and interpreted the experiments and wrote the actual paper. Mr. Leven provided useful comments.

Directed Error Detection in C++ with the Assembly-Level Model Checker StEAM [LME04] In this paper, I enhanced StEAM with the heuristic search algorithms best-first and A*. Moreover, I implemented the heuristics: *mb, int, rw, lnb, aa*, where *rw, lnb, aa* are new heuristics which I devised.

Peter Leven devised and implemented the heuristics *pba,pbb,pl1,pl2*. Also, he proposed the *lgc-Compaction*, which I implemented.

The new experiments were mainly conducted by me. Mr. Leven helped with the interpretation and contributed some scripts. The idea to illustrate the experimental results as the geometric mean of four factors was proposed by Mr. Leven, while I wrote the programs which generated the GNU-Plot diagrams from the raw experimental data.

Stefan Edelkamp invested considerable effort in making the paper sound and more readable.

I gave the presentation on the conference.

Planning in Concurrent Multiagent Systems with the Assembly-Level Model Checker StEAM [ME04] Stefan Edelkamp had the initial idea to make StEAM applicable to multi-agent systems.

The content of the paper, including the MAMP-formalism and the execution and interpretation of the experiments is mainly my contribution. Again, Stefan Edelkamp helped improving the soundness and readability of the paper.

I presented the paper at the conference.

Incremental Hashing in State Space Search [EM04] I motivated the research on incremental hashing as an important factor of StEAM's further development. Through his expertise, Stefan Edelkamp could contribute the bigger part of the formal framework. I was again responsible for the execution and interpretation of the experiments. For this, I enhanced the Atomix-solver *Atomixer* with an incremental hash function.

I gave the presentation for the paper.

Directed C++ Program Verification [Meh05] This paper gives an overview of my research. I am the only author.

Incremental Hashing for Pattern Databases [EM05a] Here, the concept of incremental hashing was transferred to the domain of action planning. Stefan Edelkamp contributed the major part of the formal framework. I enhanced the planner *mips* with an incremental hash function and executed and evaluated the experiments.

I gave the presentation for the paper.

Knowledge Acquisition and Knowledge Engineering in the ModPlan Workbench [EM05b] This paper describes the integrated planning environment *ModPlan*, which was developed in the project group 463 for which I was the co-supervisor. I was mainly responsible to help the students on practical problems (mostly in programming). I also contributed comments for the paper. Moreover, I participated in the presentation of the tool at the *Knowledge Engineering Competition* at ICAPS05.

Tutorial on Directed Model Checking [EMJ05] These are the proceedings of a tutorial at ICAPS05, which was organized by Stefan Edelkamp. Shahid Jabbar and myself contributed additional material and case studies. Moreover, I prepared and gave the talk about incremental hashing and state compaction, which constitutes about 1/5 of the overall tutorial.

Dynamic Incremental Hashing and State Reconstruction in Program Model Checking [ME05b] In this paper, incremental hashing was for the first time applied for programs model checking. I enhanced the formal framework of the previous papers with the applicability on dynamic and structured state vectors, which makes out the bigger part of the paper. Moreover, I implemented incremental hashing for the model checker StEAM. As its second contribution, the paper introduces state reconstruction, which was Stefan Edelkamp's idea in the first place. The elaboration of the concept is approximately 50% my work and 50% the work of Stefan. Again, I implemented the approach into StEAM and executed and evaluated the experiments.

The paper was meant to record our research on two novel approaches and was hence published as a technical report at the University of Dortmund.

Dynamic Incremental Hashing in Program Model Checking [ME05a] This refines the incremental hashing approach from [ME05b]. The formal framework and the overall quality of the paper was thoroughly improved by both authors. Moreover, I enhanced the introduction for a better motivation of the approach.

Bibliography

[AM02] Y. Abdeddaim and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *Workshop on Planning via Model Checking*, pages 7–13, 2002.

[AQR$^+$04a] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Exploiting program structure for model checking concurrent software. In *International Conference on Concurrency Theory (CONCUR)*, 2004.

[AQR$^+$04b] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. Technical report, Microsoft Research, 2004.

[AVL62] G. M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.

[AY98] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 175–188, 1998.

[BC01] M. Benerecetti and A. Cimatti. Symbolic model checking for multi-agent systems. In *Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, pages 312–323, 2001.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207, 1999.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Symposium on Logic in Computer Science (LICS)*, 1990.

[BCT03] P. Bertoli, A. Cimatti, and P. Traverso. Interleaving execution and planning via symbolic model checking. In *Workshop on Planning under Uncertainty and Incomplete Information*, pages 1–7, 2003.

[Ber04] Berkley Center of Electronic System Design. *BLAST User's Manual*, 2004.

[BK00] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[BR02] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Symposium on Principles of programming languages*, pages 1–3, 2002.

[Bre03] M. Brenner. Multiagent planning with partially ordered temporal plans. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1513–1514, 2003.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In *Conference on Computer-Aided Verification (CAV)*, pages 495–499, 1999.

[CEMCG$^+$02] A. Cimatti, E. Giunchiglia E. M. Clarke, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NUSMV 2: An opensource tool for symbolic model checking. In *Conference on Computer-Aided Verification (CAV)*, pages 27–31, 2002.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Transactions on Programming Languages and Systems*, volume 8, pages 244 – 263, 1986.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[Coh97] J. D. Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems*, 15(3):291–320, 1997.

[CS94] J. C. Culberson and J. Schaeffer. Efficiently searching the 15-puzzle. Technical report, University of Alberta, 1994. TR94-08.

[CS96] J. C. Culberson and J. Schaeffer. Searching with Pattern Databases. In *Canadian Conference on AI*, pages 402–416, 1996.

[CS98] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[DBL02] H. Dierks, G. Behrmann, and K. G. Lahrsen. Solving planning problems using real-time model checking. In *Workshop on Planning via Model Checking*, pages 30–39, 2002.

[DIS99] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Model Checking Software (SPIN)*, pages 261–276, 1999.

[dWTW01] M. M. de Weerdt, J. F. M. Tonino, and Cees Witteveen. Cooperative heuristic multi-agent planning. In *Belgium-Netherlands Artificial Intelligence Conference (BNAIC)*, pages 275–282, 2001.

[Ede01] S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–24, 2001.

[Ede03] S. Edelkamp. Promela planning. In *Workshop on Model Checking Software (SPIN)*, pages 197–212, 2003.

[EJS04] S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *German Conference on Artificial Intelligence (KI)*, pages 226–240, 2004.

[ELL01] S. Edelkamp and A. Lluch-Lafuente. *HSF-Spin User Manual*, 2001.

[ELL04] S. Edelkamp and A. Lluch-Lafuente. Abstraction in directed model checking. In *Workshop on Connecting Planning Theory with Practice*, pages 7–13, 2004.

[ELLL01] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Workshop on Model Checking Software (SPIN)*, pages 57–79, 2001.

[ELLL04] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2-3):247–267, 2004.

[EM03] S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.

[EM04] S. Edelkamp and T. Mehler. Incremental hashing in state space search. In *Workshop on New Results in Planning, Scheduling and Design (PUK)*, pages 15–29, 2004.

[EM05a] S. Edelkamp and T. Mehler. Incremental hashing for pattern databases. In *Poster Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, pages 17–20, 2005.

[EM05b] S. Edelkamp and T. Mehler. Knowledge acquisition and knowledge engineering in the ModPlan workbench. In *International Competition on Knowledge Engineering for Planning and Scheduling*, pages 26–33, 2005.

[EMJ05] S. Edelkamp, T. Mehler, and S. Jabbar, editors. *ICAPS Tutorial on Directed Model Checking*, 2005.

[EPP05] S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *International Workshop on Model Checking Software (SPIN)*, pages 43–57, 2005.

[FGK97] L. Fredlund, J. F. Groote, and H. Korver. Formal Verification of a Leader Election Protocol in Process Algebra. *Theoretical Computer Science*, 177(2):459–486, 1997.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs of infinite systems with PVS. In *Conference on Computer-Aided Verification (CAV)*, pages 72–83, 1997.

[GV02] 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.

[GV03] A. Groce and W. Visser. What went wrong: Explaining counter examples. In *Workshop on Model Checking of Software (SPIN)*, pages 121–135, 2003.

[Hav99] K. Havelund. Java PathFinder user guide. Technical report, NASA Ames Research Center, 1999.

[HBG05] P. Haslum, B. Bonet, and H. Geffner. New admissible heuristics for domain-independent planning. In *National Conference on Artificial Intelligence (AAAI)*, pages 1163–1168, 2005.

[HEFN01] F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to atomix. In *German Conference on Artificial Intelligence (KI)*, pages 229–243, 2001.

[Hel04] M. Helmert. A planning heuristic based on causal graph analysis. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 161–170, 2004.

[HJMS03] A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Workshop on Model Checking Software (SPIN)*, pages 235–239, 2003.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hol96] G. J. Holzmann. An analysis of bistate hashing. In *International Symposium on Protocol Specification, Testing and Verification*, pages 301–314. Chapman & Hall, Ltd., 1996.

[Hol97a] G. J. Holzmann. Designing bug-free protocols with SPIN. *Computer Communications Journal*, 20(2):97–105, 1997.

[Hol97b] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[Hol97c] G. J. Holzmann. State compression in SPIN. In *Third Spin Workshop*, Twente University, The Netherlands, 1997.

[Hol98] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.

[Hol03] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HS01] M. Holzer and S. Schwoon. Assembling molecules in Atomix is hard. Technical Report 101, Institut für Informatik, Technische Universität München, 2001.

[HSAHB99] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Conference on Uncertainty in Articial Intelligence (UAI)*, pages 279–288, 1999.

[HT99] J. Hatcliff and O. Tkachuck. The Bandera Tools for Model-checking Java Source Code: A User's Manual. Technical report, Kansas State University, 1999.

[HZ01] E. A. Hansen and S. Zilberstein. LAO * : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[IDS98] R. Iosif, C. Demartini, and R. Sisto. Modeling and validation of Java multithreading applications using SPIN. In *Workshop on automata theoretic verification with SPIN*, pages 5–19, 1998.

[Ios01] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *International Conference on Automated Software Engineering (ICSE)*, pages 26–29, 2001.

[JE05] S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 313–329, 2005.

[KDH00] J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and ressources. In *European Conference on Artificial Intelligence (ECAI)*, pages 501–505, 2000.

[Knu98] D. Knuth. *The Art of Computer Progarmming - Vol.3: Sorting and Searching (2nd edition)*. Addison-Wesley, 1998.

[Kor85] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor03] R. E. Korf. Breadth-first frontier search with delayed duplicate detection. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 87–92, 2003.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[KS05] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1380–1386, 2005.

[KZ00] R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.

[Lan96] H. Landman. *Games of No Chance*, chapter Eyespace values in Go, pages 227–257. Cambridge University Press, 1996.

[Leh49] H. D. Lehmer. Mathematical methods in large-scale computing units. In *Symposium on Large-Scale Digital Calculating Machienery*, pages 141–146. Cambridge, Massachusetts, Harvard University Press, 1949.

[LL03] A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *Workshop on Model Checking and Artificial Intelligence (MoChart)*, pages 77–86, 2003.

[LME04] P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Workshop on Model Checking Software (SPIN)*, pages 39–56, 2004.

[LMS02] F. Laroussinie, N. Markey, and Ph. Schnoebelen. On model checking durational kripke structures. In *International Conference on Foundations of Software Science and Computation Structures*, pages 264–279, 2002.

[LV01] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Workshop on Model Checking Software (SPIN)*, pages 80–94, 2001.

[Mar91] T. A. Marsland. *Encyclopedia of Artificial Intelligence*, chapter Computer Chess and Search, pages 224–241. J. Wiley & Sons, 1991.

[McM92] K. L. McMillan. The SMV system. Technical report, Carnegie-Mellon University, 1992.

[MD05] M. Madanlal and D. L. Dill. An incremental heap canonicalization algorithm. In *Workshop on Model Checking Software (SPIN)*, pages 28–42, 2005.

[ME04] T. Mehler and S. Edelkamp. Planning in concurrent multiagent systems with the assembly model checker StEAM. In *Poster Procedings of German Conference on Artificial Intelligence (KI)*, pages 16–30, 2004.

[ME05a] T. Mehler and S. Edelkamp. Dynamic incremental hashing in program model checking. *Electronic Notes on Theoretical Computer Science (ENTCS)*, 149(2):51–69, 2005.

[ME05b] T. Mehler and S. Edelkamp. *Dynamic Incremental Hashing and State Reconstruction in Program Model Checking*. Technical report, University of Dortmund, 2005.

[Meh02] T. Mehler. Gerichtete Java-Programmvalidation. Master's thesis, Institute for Computer Science, University of Freiburg, 2002.

[Meh05] T. Mehler. Directed C++ program verification. In *International Conference on Automated Planning and Scheduling (ICAPS) - Doctoral Consortium*, pages 58–61, 2005.

[MJ05] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *Workshop on Model Checking Software (SPIN)*, pages 251–265, 2005.

[ML03] T. Mehler and P. Leven. Introduction to StEAM - an assembly-level software model checker. Technical report, University of Freiburg, 2003.

[MPC$^+$02] 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.

[Pun77] A. Puneli. The temporal logic of programs. In *Symposium on foundations of computer science (FOCS)*, pages 46–57, 1977.

[PvV$^+$02] M. Pěchouček, A. Říha, J. Vokřínek, V. Mařík, and V. Pražma. Explantech: applying multi-agent systems in production planning. *International Journal of Production Research*, 40(15):3681–3692, 2002.

[QN04] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Symposium on Theoretical Aspects of Computer Science (TACAS)*, pages 497–511, 2004.

[RM94] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[RN95] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach*. Prentice Hall, 1995.

[Sch98] U. Schoening. *Algorithmen - kurzgefasst, Kapitel: Algebraische und Zahlentheoretische Algorithmen.* Spektrum Verlag, 1998.

[SD96] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90, 1996.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992.

[TK93] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 756–761, 1993.

[VHBP00a] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. In *Workshop on Advances in Verification, (WAVe)*, pages 28–34, 2000.

[VHBP00b] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *International Conference on Automated Software Engineering (ICSE)*, pages 3–12, 2000.

[Zob70] A. L. Zobrist. A new hashing method with application for game playing. Technical report, University of Wisconsin, Computer Science Department, March 1970.

Index