Universität Passau
Fakultät für Informatik und Mathematik

Verification Tasks for Software Model Checking

Specification and Presentation of Verification Tasks in CPAchecker

Master Thesis

Author:
Alexander von Rhein

Supervisor:
Prof. Dr. Dirk Beyer

August 23, 2010

von Rhein, Alexander:
Verification Tasks for Software Model Checking
Master Thesis, University of Passau, 2010.

Abstract

Software model checking is an automatic method for formal verification of software programs, that can construct a certificate of correctness or an error path as witness for a bug. Because of the dramatical growth of size and complexity of software programs and the growing dependency on software products, it is especially important to ensure reliability of these products. While software model checking has been successful in discovering subtle bugs in code, it is still quite cumbersome to use. This work introduces the concept of *Verification Tasks* to simplify and organize the user interface. Furthermore we develop a new specification language and a new graphical user interface for the verification tool CPAchecker. The user interface is implemented as the Eclipse-plugin *CPAclipse*, which is publicly available.

Contents

List of Figures

Chapter 1

Introduction

Software is probably the most omnipresent human-made material in our modern society. It has become a mission-critical part in many aspects of daily life like telecommunications, airplanes, automobiles, entertainment and so on. There are technologies that are extremely expensive (space travel) or dangerous for human life (transportation safety, medicine). Especially these technologies demand high-quality software to ensure reliability. Previous software disasters have endangered human life or lead to severe loss of reputation and money [Neu09].

The ever-growing size and complexity of computer programs makes the task of ensuring the quality and reliability of software dramatically difficult. This lead to the research field of software reliability engineering [Lyu07] which has identified *Fault removal* as one major technique to achieve reliable software systems. Fault removal uses verification to detect the existence of faults and eliminate them.

This thesis concentrates on one approach to verification called *software model checking*. Model checking uses algorithmic analyses to prove properties of the program. The user has to formalize the properties in a formal language given by the model checking tool. The tool then proves whether the properties are fulfilled by the program. This is a rather costly approach to ensure reliability because it requires exhaustive exploration of all possible program behaviors. If the relatively high cost is invested and the analysis terminates with a positive result the user receives the ultimate certificate of correctness. The tool has proved that every possible program behavior fulfills the specification.

One of the remaining problems is that the user has to provide the specification. If this specification does contain errors the model checking result leads to wrong results and the error might only be detected in later steps of the software engineering process which leads to significant cost increases. This thesis introduces a new language for specification of program properties and a user interface for a software model checking tool. The user interface is developed as the plugin CPAclipse for the integrated development environment Eclipse[1] which is available on an eclipse update site[2]. The plugin is developed for the Eclipse version 3.5.2 using Java version 1.6 but it should run on later versions as well. In addition to this the *verification task* is introduced. A verification task contains any information needed to start a verification process and the information produced by this process. Verification tasks provide a simple interface to the programmer. The

[1] `www.eclipse.org`

[2] `http://www.sosy-lab.org/~dbeyer/eclipse-download/`

verification is executed completely automatic and requires no further knowledge of model checking or formal notation.

In verification tasks, the different concerns of implementation and specification are separated. This allows separating the development of these concerns on different development teams. Because of psychological reasons this separation can result in more exact specifications and in more reliable programs.

The software model checking tool CPAchecker that is extended during the work for this thesis is a flexible, extensible tool for verification of C-programs. CPAchecker is based on the formal framework described in [BHT07] and [BHT08]. The CPAchecker distribution contains pre-written specifications for simple, generic specification cases like the reachability of null pointer dereferences. More complex, domain-specific specifications can easily be developed with the support of CPAclipse.

1.1 Outline of the Thesis

This thesis consists of three main chapters and a background chapter. In the background chapter (Chapter 2) we describe various topics that are needed in the other chapters. The topics are Software Model Checking in general, the Software Model Checking tool CPAchecker, a method to specify program paths and a short introduction to incremental software verification. Chapter 3 introduces the new concept of Verification Tasks for Software Model Checking. A part of each Verification Task is the set of input components like the program source and the specification of what is to be verified. In Chapter 4 a new language for definition of these specifications is developed. This language is also integrated into the existing tool CPAchecker. Chapter 5 describes how the Concept of Verification Tasks is integrated into the user interface of CPAchecker. This user interface is implemented in the practical work for this thesis as an Eclipse-plugin.

Chapter 2

Background

2.1 Software Model Checking

Software model checking is the algorithmic analysis of programs to prove properties of their executions [JM09]. In model checking, *models* of the program are verified according to a given *specification*. Both, the model and the specification are formulated in precise formal languages. Formally the problem of model checking can be expressed as follows. Given a desired specification S and a model M decide if $M \models S$. Because the decision process involves checking all possible behaviors (that affect the specification) of the model, it is normally executed by a model checking tool (*model checker*).

Compared with other methods of finding errors, software model checking has some distinct advantages. The principal validation methods for complex software systems are simulation, testing, deductive verification and model checking [CGP99]. Testing and simulation are conducted by performing experiments on the actual product (testing) or a model of the product (simulation). In both cases the test subject is controlled by the test specification and the resulting behavior is audited. Deductive verification uses axioms and proof rules to prove the correctness of systems. As this involves manual construction of proofs, the method is time intensive and error prone. Testing and simulation are cost-effective ways to find errors fast when the code still contains many errors. This is normally the case in earlier phases of a software project. But when the development process advances and only few errors remain the efficiency of this method decreases. The problem of testing and simulation is that one test can only verify one execution path.

> *Program testing can be used to show the presence of bugs, but never to show their absence! [DDH72]*

The advantages of model checking are:

- Model checking conducts an exhaustive exploration of all possible behaviors of the model. In contrast to testing and simulation all cases are considered and not just one.

- It is fully automatic and its execution requires no user interaction or supervision during the execution process.

- If the verification fails the model checking tool produces a *counterexample* that shows which erroneous behavior caused the failure. This faulty trace provides a priceless insight to understanding the real reason for the failure as well as important clues for fixing the problem [CGP99].

Disadvantages of model checking are that it is generally more complex to define a specification for model checking than defining a test case (for similar problems). This is because specification languages must provide an ability to define expected results for different program execution paths. In contrast to this a test case must only consider one program execution path. The main disadvantage of model checking is the *state explosion* that can occur if the system being verified has statements which cannot be evaluated by the model checker. These statements might depend on source code that is not available or on user input. The model checker has to generate states for every possible outcome of the statement and continue with these states.

Model checking consists of several tasks which are explained in the following listing [CGP99]:

1. *Modeling*: In the first task the subject of the test (typically a source code) is converted into a formal model that is accepted by the model checker. If the model checker accepts the source code language as it is, no conversion is necessary.

2. *Specification*: Before the verification can start the specification must be defined. A possible though complex specification language is temporal logic.

   An important issue in specification is *completeness* [CGP99]. Model checkers can prove that a model satisfies the specification. But it is impossible to determine whether all requirements are covered by the specification.

3. *Verification*: The verification process itself is executed completely automatic. However human assistance is needed to analyze the result of the verification. If the verification fails it produces an error trace that helps the designer in tracking down where the error occurred. Error traces might also result from incorrect modeling, incorrect specification or an incorrect implementation of the verification algorithm. In each of these cases the error trace will be a *false negative* result and human interaction will be necessary to detect these cases.

2.2 CPAchecker

CPAchecker is a tool and a framework that aims at easy integration of new software verification components [BK09]. CPAchecker uses the concept of *configurable program analysis* (CPA) which is explained in Subsection 2.2.2. This framework allows CPAchecker to be used as a precise model checker (Section 2.1) as well as an efficient program analyzer [BHT07]. The central data structure in CPAchecker is the control-flow automaton which is defined in Subsection 2.2.1.

The input language of CPAchecker is the "C Intermediate Language" (CIL) [NMRW02]. This language allows a subset of the constructs of C and comes with a tool that translates all possible C-programs into CIL-programs. The tool breaks down certain complicated C constructs into simpler ones. The conversion from source programs written in C to CIL is the first step in the modeling task as described above. The simplified language allows the developers of configurable program analyses to concentrate on a relatively small number of language constructs.

The specification defines which program behaviors have to be found and identified as errors. In CPAchecker this can be expressed in the CPAchecker specification language (Chapter 4). It allows defining automata which analyze the program and can signal when an unwanted behavior is detected.

An interesting aspect in the concept of CPAchecker is that users can define new configurable program analyses and use them in combination with the existing predefined analyses. To define a new analysis the user must implement a CPA as defined by Beyer [BHT07] and described in Section 2.2.2.

In this master thesis we developed an Eclipse [1] plugin that provides a convenient user interface for CPAchecker. The plugin is described in Chapter 5. It is provided in addition to the already existing command line user-interface.

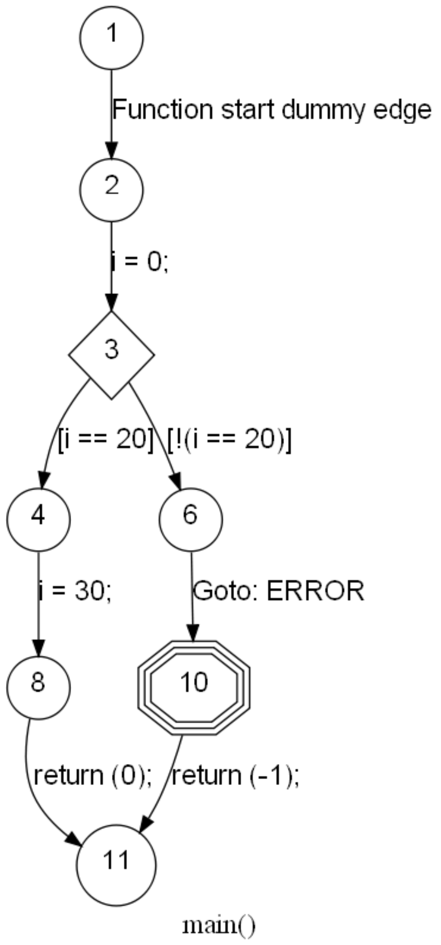
2.2.1 Programs and Control-Flow Automata

The internal representation of the model (the program to be verified) in CPAchecker is a *control-flow automaton* (CFA). This definition of control-flow automata is taken from Beyer [BCG$^+$09] to be consistent with the definition in CPAchecker.

A CFA $A = (L, G)$ consists of a set L of program locations, which model the program counter l, and a set $G \subseteq L \times \mathit{Ops} \times L$ of control flow edges, which model the operations that are executed when the control flows from one program location to another. The set of variables that occur in operations from *Ops* is denoted by X. A *program* $P = (A, l_0, l_E)$ consists of a CFA $A = (L, G)$ (which models the control flow of the program), an initial program location $l_0 \in L$ (which models the program entry) such that G does not contain any edge $(\cdot, \cdot, l_0)$, and a target program location $l_E \in L$ (which models the error location). Further definitions that are used in this thesis will only be given informally here. A *program path* is a sequence of pairs of operations and locations that represents a syntactical walk through the CFA. A *feasible* program path is a path that is not only syntactically but also semantically correct. Informally described a feasible program path is a path where each state on the path is produced from its predecessor by modifying it according to the operation on the edge between them. So a feasible path is a path that could be followed by the program when it is executed. The CPA algorithm computes abstractions during the analysis

[1] `www.eclipse.org`

```c
int main() {
   i = 0;
   if(i == 20){
      i = 30;
   } else {
      goto ERROR;
   }
   return (0);
   ERROR:
   return (-1);
}
```



Source Code A

Control-Flow Automaton for A

Figure 2.1: Example for a Control-Flow Automaton

which allows for a more efficient verification run but introduce false negatives. These false errors are program paths that are not feasible.

CPAchecker uses the parser from the Eclipse CDT plugin to generate CFAs. An example of a generated CFA is displayed in Figure 2.1. Due to the size of the graph that is generated from such a small example we could not include larger examples.

2.2.2 Configurable Program Analysis

This section describes the concept of *configurable program analysis* (CPA) [BHT07] and *configurable program analysis with precision adjustment* (CPA+) [BHT08] and an algorithm that uses the CPA+ for reachability analyses. Parts of this section are cited from these sources. In most parts of the thesis the additional features of the CPA+ are not needed, so in most cases the simpler CPA from [BHT07] will be used. This is valid because every CPA can be used as CPA+ with a default implementation of the additional features.

Automatic program verification requires a choice between precision and efficiency. The more precise a method, the fewer false positives it will produce, but also the more expensive it is, and thus applicable to fewer programs. Historically, this trade-off was reflected in two major approaches to static verification: program analysis and model checking. Program analysis makes efficient but more inaccurate analyses that might for example lose precision at the join points of program paths. Model checking explores an abstract reachability tree that keeps separate program paths separate and thus makes the analysis expensive. The concept of CPA+ covers both approaches and thus can be used by either of the two analysis approaches.

A configurable program analysis with precision adjustment

$$\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathit{merge}, \mathit{stop}, \mathit{prec})$$

consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, a merge operator *merge*, a termination check *stop* and a precision adjustment function *prec*. These six components define the behavior of the CPA+ and therefore the precision and cost of the verification. Most operators in the concept are parametric to an abstract state with the precision that was used to compute this state. We call this an *abstract state with precision.* The abstract domain $D = (C, \varepsilon, [\![\cdot]\!])$ models the abstraction of the source code program that this CPA maintains. D consists of a set of concrete states C, a semi-lattice $\varepsilon = (E, \top, \bot, \sqsubseteq, \sqcup)$ and a concretization function $[\![\cdot]\!]$. The semi-lattice contains the set of abstract domain elements E, the top and bottom elements $\top \in E$ and $\bot \in E$, the partial order $\sqsubseteq \subseteq E \times E$ and the join function $\sqcup \subseteq E \times E \to E$. The set of precisions Π determines the possible precisions of the abstract domain. The transfer relation $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ (where G is the set of control flow edges) assigns to each abstract state e of the lattice and each control flow edge g a set of abstract states with precision that are successors of the state e if the g is taken. The merge operator $\mathit{merge} : E \times E \times \Pi \to E$ combines the information of two abstract states with precision. This operator can be used to reduce the number of abstract states and thereby the complexity of the analysis. If the operator is used it might also decrease the precision of the analysis because some information is lost. The termination check $\mathit{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks if a given abstract state with the given precision is already in the reached set of abstract states. If the state is already in the reached set, it is already explored and the current path needs not be explored further. The precision adjustment function computes a new abstract state e' with precision p' from an abstract state e with precision p using the information of all abstract states with precision that are currently stored in the reached set. For this thesis the precision adjustment function has to be extended to the following signature: $\mathit{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi \times B$ where the elements of the set $B = \{\mathit{break}, \mathit{continue}\}$ determine the further process of the analysis. For example if an CPA+ has found an error its precision adjustment function would signal *break* and the algorithm would terminate. Any further details on the formal background of CPAs can be found in [BHT08].

Configurable program analyses can be used separately but most of the time several CPAs are combined in one verification process. This combination can

Algorithm 2.2.1 *CPA+*$(\mathbb{D}, R_0, W_0)$

```
Input: a configurable program analysis with dynamic precision
       adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$,
       a set of reachable abstract states with precisions $R_0 \subseteq E \times \Pi$,
       a subset $W_0 \subseteq R_0$ of frontier abstract states,
       where $E$ denotes the set of elements of the semi-lattice $D$
Output: a set of reachable abstract states with precision,
        a subset of frontier abstract states with precision
Variables: a set $reached$ of elements of $E \times \Pi$,
           a set $wait$ of elements of $E \times \Pi$
  $reached := R_0$;
  $wait := W_0$;
  while $wait \neq \emptyset$ do
    pop $(e, \pi)$ from $wait$;
    for each $\hat{e}$ with $e \rightsquigarrow (\hat{e}, \pi)$ do
      // Adjust the precision.
      $(e', \pi', s) = prec(\hat{e}, \pi, reached)$;
      for each $(e'', \pi'') \in reached$ do
        // Combine with existing abstract state.
        $e_{new} := merge(e', e'', \pi')$;
        if $e_{new} \neq e''$ then
          $wait := (wait \cup \{(e_{new}, \pi')\}) \setminus \{(e'', \pi'')\}$;
          $reached := (reached \cup \{(e_{new}, \pi')\}) \setminus \{(e'', \pi'')\}$;
      // Add new abstract state?
      if $\neg\ stop(e', \{e \mid (e, \cdot) \in reached\}, \pi')$ then
        $wait := wait \cup \{(e', \pi')\}$;
        $reached := reached \cup \{(e', \pi')\}$;
        if $s = break$ then
          $wait := wait \cup \{(e, \pi)\}$;
          $reached := reached \cup \{(e, \pi)\}$;
          return $(reached, wait)$
  return $(reached, \emptyset)$
```

be accomplished with the *composite program analysis*. The composite program analysis is a CPA that can be composed of several CPAs. CPAs used in the same CompositeCPA can even interact and share information. A formal definition of this CPA was give by Beyer [BHT08]. We call CPAs that can contain other CPAs *Wrapper CPA*.

Because Wrapper CPAs implement the CPA interface themselves, Wrapper CPAs can be nested. Therefore the user can create tree-structures where the inner nodes are Wrapper CPAs and the leaves are normal CPAs. This possibility has to be considered when working with user defined CPA-configurations.

The concept of configurable program analysis with precision adjustment is now used in an algorithm to perform reachability analyses. The Algorithm 2.2.1 takes as input a CPA+, a set of abstract states with precision that are already marked as reached and a list of abstract states with precision which successors are still to be explored. The algorithm returns if either no more states can be explored or a state with the precision *break* is found. The return values of the algorithm are the two sets of abstract states (which are modified). The algorithm shall be started with the initial sets $reached_0 = wait_0 = (e_0, \pi_0)$ where e_0 is the initial abstract state of the analysis and π_0 is the initial precision. The algorithm keeps updating two sets of abstract states with precision: a list *reached* to store all abstract states with precision that are found to be reachable, and a set *wait* to store the reachable states which successors were not yet processed.

The state exploration starts from the initial state e_0 with initial precision π_0. For a current abstract state e with precision π the algorithm first considers each successor e' with the new precision π', according to the transfer relation. For every successor with precision the *prec* function is used to adjust the precision of the algorithm and to determine if the algorithm should stop after exploring this state. Now, using the merge operator, the abstract successor state is combined with each state in the reached set. If the *stop* operator determines that the abstract element e' with precision π' is already contained in the reached set the analysis of this element the current path needs not be explored further. Otherwise it is added to the sets *reached* and *wait*. If the precision adjustment function stated that the algorithm should terminate after exploring the current successor state ($s = break$), the original state is added to the waitlist and the reached set. Then both sets are returned to the caller for further processing. It is necessary to re-add the original state to the waitlist because there might still be unexplored successors of this state that have to be considered in further analyses. When the waitlist is empty all reachable abstract states have been explored and the algorithm terminates.

2.3 Identification of Program Paths using Automata

This section describes how automata can be used to identify program paths. We develop a language for the definition of these automata in Chapter 4 Specification Language. The automata can be used to specify a set of program paths (as errors) and let the model-checking tool determine if these paths occur in the concrete program.

The representation of the program is given by a control-flow automaton $C = (L, G, l_0)$ where L is the set of program locations, $G \subseteq L \times Ops \times L$ is the set of transitions between program locations, Ops is the set of operations and l_0 is the initial program location. The set Ops includes a special operation *end* which represents the last operation of a program run. A program path is an ordered list of tupels of program locations and operations (i.e., $path_0 = ((l_0, op_1), (l_1, op_2), ...)$ where $L_0 \in L$ and $op_1 \in Ops$). The length of the path might be infinite, e.g due to loops or non-terminating recursion (halting-problem).

An automaton $\mathbb{A} = (Q, \Sigma, q_0, q_t, \rightarrow)$ consists of the set of states Q, the set of input symbols $\Sigma \subseteq G$, the initial state q_0, a special state q_t and the relation $\rightarrow \subseteq Q \times \Sigma \times Q$ which determines the successor states. The automata are used to specify paths to target states so the only accepting state of the automata is q_t

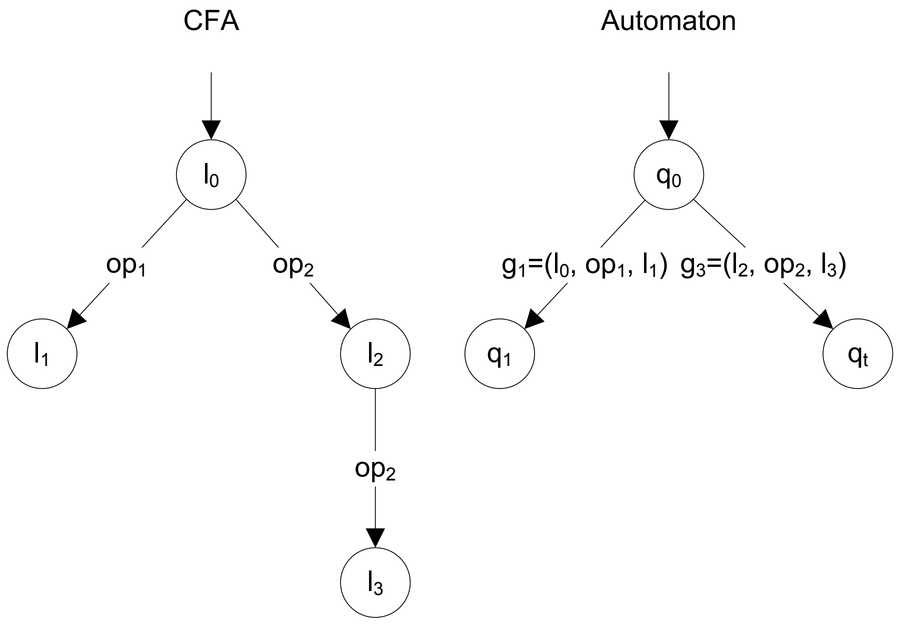


Figure 2.2: Abstract Automaton Example

(this is the only target state). Once this accepting state is reached, the analysis is terminated.

During the exploration of the CFA the automaton is evaluated in parallel. This means they both start in their initial states and for each transition taken by the CFA, the automaton takes one transition if the CFA-transition is a valid input symbol for the current automaton state.

The usage of the automata is shown in the following simple example (see Figure 2.2). Consider the following example CFA $C = (L_C, G_C, l_0)$ with the set of operations $Ops = \{op_1, op_2\}$, the set of program locations $L_C = \{l_0, l_1, l_2, l_3\}$ and the set of transitions $G_C = \{g_1, g_2, g_3\}$ with $g_1 = (l_0, op_1, l_1)$, $g_2 = (l_0, op_2, l_2)$ and $g_3 = (l_2, op_2, l_3)$. This CFA C is verified with the automaton $A = (Q_A, \Sigma_A, q_0, q_t, \rightarrow_A)$ with $Q_A = \{q_0, q_1, q_t\}$, $\Sigma_A = \{g_1, g_3\}$ and $\rightarrow_A = \{(q_0, g_1, q_1), (q_0, g_3, q_t)\}$. The concrete CFA yields the two program paths $p_1 = ((l_0, op_1), (l_1, end))$ and $p_2 = ((l_0, op_2), (l_2, op_2), (l_3, end))$.

During the analysis of the program path p_1 the automaton reaches the automaton state q_1 after the first step. Because no further transition is taken the path p_1 is not identified as error. In the program path p_2, the first element of the path is (l_0, op_2). The relation $\rightarrow$ does not contain a tuple for this input symbol so the automaton stays in the current state[2]. The next element of the path is (l_2, op_2) and the current state of the automaton is still q_0. The relation $\rightarrow$ does contain a tuple for this combination: (q_0, g_3, q_t). Therefore the successor state of the automaton is q_t and the program path p_2 is accepted by the automaton. This

[2] This is a derivation from the normal automaton definition.

marks the path as a path to a state of interest in the program (in most cases this will be an error).

2.4 Incremental Software Verification

There are several interpretations of what incremental software verification means. This section presents three approaches that aim to reduce the complexity of the verification task by reusing results of previous tasks. The difference between the approaches is what is seen as an *increment*.

In feature-oriented product lines a software product consists of a set of features F. Developers build programs by incrementally linking features. In *incremental aspect model checking* [KF07] an increment is a feature. When a product is verified, information that has been gathered during previous verifications of subsets of F can be reused.

Another approach is to use the traditional decomposition of software programs into modular components and verify components independently. In *modular verification* [GL94] the model checker verifies components. Because at each time only some modules are considered, this reduces the complexity of the analysis. Relations between the component and the system containing the component are used during the verification process.

In incremental software development code is developed and evolves over time. During development of the application test cases and specifications are written and verified. Because the same code is rewritten and extended old specifications have to be re-verified when they might be affected by code changes. The idea behind the approach of *incremental software verification* [BHJM04] is that computational effort should only be spent for changed parts of the program. This requires an analysis of the changed program parts and of the specifications. If the CPA interface as described in Section 2.2.2 is used, this analysis also depends on which CPAs are used in the current configuration and which CPAs were used previously to compute the known information.

Common to all these approaches is that information computed in previous iterations is saved for future reuse. When the verification tool is run again it has to consider which of the saved information can be reused and which must be recomputed.

Chapter 3

Verification Tasks

This chapter introduces the concept of verification tasks. The introduction is given on the basis of general software verification tools. If suitable, examples from the concrete tool CPAchecker are given as illustration. Most of the aspects of verification tasks described in this section have been implemented in CPAchecker.

Verification tasks are designed to contain all information that is needed for the verification process. This avoids dependencies between the tasks and helps to provide better usability.

The Section 3.1 describes which elements compose a verification task. In the following Sections (3.2 to 3.6) each of the components are described more detailed. The section about verification task collections (Section 3.7) describes an approach to structure verification tasks. Section 3.8 describes how the concept of verification tasks is integrated into CPAchecker.

3.1 Components

Every software model verification tool has a defined set of input and output information. A verification task encapsulates the input information and the corresponding output information. The input information configures the tool and states which file is to be verified on which specification. As software model verification tools are defined to be non-interactive, all input information is known when the tool is started. Typically this information is passed on the command-line or in a file that is referenced in a command-line option. The output information contains the primary verification result and other information like details of encountered errors. The primary verification result states if the verified file was found to be fulfilling the specification or not. This leads to a three-state type for the primary verification result: $\{Error\_Found, No\_Error\_Found, Unknown\}$. The Unknown state would be used if the tool cannot determine the result. This might be the case if an internal error occurs or invalid input information is given.

The *input information* consists of four different parts:

1. The source code that is to be verified
2. The specification defining the behavior that is to be verified
3. An optional set of configuration options

4. A set of results (parts of the output information) from previous analyses that can be reused

Informally a specification defines which program behavior is considered an error. A configuration contains any other settings used by the verification tool. To provide a convenient user interface it is necessary that the three input components of a verification task are stored and edited separately. This makes the verification tool easier to use for non-experts and supports reuse of the components for other verification tasks and further code development. The approach keeps program properties separate from the source code and makes it easier to maintain both the specification and the source code [KLM+97].

Each part of verification tasks will be discussed in the following sections.

3.2 Source code

The source code input of software verification tools can be basically in any language and any level of abstraction. Of course typical software verification tools allow only a certain programming language or a small set of programming languages as input. The level of abstraction needs to be as low (as near to the actual programming language) as it is necessary to make meaningful specification statements about code elements. During abstraction the source code representation loses information. If this information is needed to declare what the tool should verify, the abstraction level is too high. Consequently the source code input language of most software verification tools is the programming language itself or a simplified version of the language.

For example CPAchecker accepts only C-programs in “C Intermediate Language” (CIL) [NMRW02] as described in Section 2.2. Because CPAchecker allows only CIL-preprocessed source code it is able to concentrate on fewer source language constructs. But using preprocessed source code has some disadvantages concerning the user interaction. Firstly, the user has to do the preprocessing manually if the tool does not support this in a step before starting the verification. Secondly, it introduces a dependency from the verification tool to the preprocessor. If a user wants to use the verification tool he must install the preprocessor which might introduce portability problems. Thirdly, preprocessors change the code and line numbering. So when the tool issues a message concerning a certain statement or code line it always references the preprocessed code. The statement might look totally different or might be in another line in the original source code. For a research project it is totally fine to use a preprocessor like CIL because the research to be performed is more important than the ease of installation or the accuracy of user messages.

3.3 Specification

The specification allows the user to define what behavior should be considered erroneous during the verification run. Informal specifications might be very simple like “The program must not dereference null pointers.” But there are also more complex specifications. For example a lock manager that allows only alternating calls to the functions `lock()` and `unlock()` requires a specification-automaton that simulates this behavior. An example for such an automaton in the CPAchecker specification language is given in Section 4.3.2. There are some specifications that can be pre-defined by the tool developers (like “no null pointer dereferencing”) because probably every user will want to detect these common errors. Some specifications (like the locking example) are so domain specific that they can only be written by a domain expert. The domain expert would probably like to merge the complex domain-specific specifications with the simple pre-defined specifications.

This examples leads to two requirements for specifications:

1. The specification must allow multiple levels of detail. Simple specifications must be simple to define. There must be a way to define more complex specifications (like automata).

2. The user must be able to merge specifications into specification bundles.

To distinguish specifications from other user defined options this work uses the following definition.

Definition 3.1 (Specification) *A specification defines a (possibly infinite) set of program execution paths that are considered to be errors. The verification has to terminate with an $Error\_Found$ or an $Unknown$ result if an error is reachable.*

This definition is problematic when the evaluation of a specification depends on certain configuration options, or if configuration options influence the tool’s capability to evaluate a specification. The configuration option might be seen as a specification option in this case. More discussion on the problem of distinguishing specification options from configuration options is given in Section 3.4.

The definition also introduces a new aspect that errors are program paths. In previous works the error in a program was defined by an error location, that is by one statement or label in the code. In this definition an error is a whole program path (or a set of similar program paths). The difference is that, in the new definition, a program path $(l_1, l_2, ..., l_n)$ might be identified as error, but the program path $(l'_1, l'_2, ..., l_n)$ which ends in the same location might be no error. This means the new error depends not only on the current location (whether this is an error location or not) but on arbitrary properties of the current program path. These properties can be expressed by the use of automata that observe the program path.

CPAchecker provides a language to define specifications according to the requirements developed above. This specification language of CPAchecker is described in Chapter 4.

3.4 Configuration

In order to separate the concerns of specification and configuration we define a separate language for each concern. However it turned out that there is no clear classification between the concepts of specification and configuration. In our opinion the specification is a special configuration option that configures the analysis algorithm. Depending on this configuration option the algorithm decides which program paths are considered errors. Furthermore in CPAchecker there are dependencies from the specification on other configuration options. Especially important is the option “cpas” of CPAchecker that configures which analyses are used by the tool. Certain specifications can only be evaluated if a special analysis is used. So the specification depends on the correct setting of this configuration option. One might say that the option itself is a part of the specification because of this dependency. On the other hand the option definitely configures the tool’s algorithm and does not *define program execution paths that are considered to be errors*. Therefore we consider the “cpas” option and the specification to be separate configuration aspects.

It is desirable that the user does not need to write a configuration file for simple, common verification scenarios. CPAchecker uses two methods to accomplish this:

1. Default specifications for simple or common tasks are provided with corresponding configurations

2. For all configuration options CPAchecker assumes default values

The most basic verification scenarios are covered by specifications that are provided with the CPAchecker implementation. These simple specifications come with configuration files that contain options required by the specification. This aspect primarily concerns the “cpas” option mentioned above. For example the informal specification “No null pointer dereferences” depends on the analysis of pointers. If this specification is used, the configuration file must contain the information that the pointer analysis must be used.

In CPAchecker the configuration options are set as key-value pairs in a Java Properties File[1].

3.5 Output information

Each run of the verification task produces a set of output information. Some of this information is directly displayed to the user and some is saved to files for later inspection. Which output information is generated depends on the used analyses and on the verification result. In the following paragraphs some types of output information are described.

[1] The Properties File format is defined in the Javadoc of the Java Properties class.

Verification Result

The most important result is whether the verification has been completed and if it has found an error or not. This information is encoded in the three states $\{Error\_Found, No\_Error\_Found, Unknown\}$. In the CPAchecker command line interface this information is displayed on the terminal together with statistical information like timing and the size of the set of reachable locations. In the Eclipse-Plugin the information is displayed as icons in front of the name of the task. More information on the Eclipse-Plugin is given in Chapter 5.

Log

Another important result is the log, where messages on the process of the analysis are accumulated. CPAchecker generates two versions of the log. The levels of detail of these logs can be configured separately. The first version is printed on the terminal and the second version is saved to a file. As described in Section 4.3.6 the automata can print user defined messages to the log. In addition to that the log contains debugging information from the used CPAs and general information on the verification progress. This makes the log one of the most information intensive and certainly the most user configurable output medium. But for most users the log will contain too much information, and it is not easy to extract the relevant information due to missing structures. Users that are not using CPAchecker on a regular basis, or that use only simple specifications probably want an easier log representation.

In a graphical user interface environment this could be a log with the ability to set a filter. The user could filter information that he is not interested in and would see only the important results. Single log messages could be highlighted according to their importance. The log messages could contain meta-information like (if applicable) which source code element the message is about. If this meta-information is present the user interface can provide a link to the source code and thereby improve the usability of verification results.

However some output information is too large to be encoded in a single log message, or contains inherent structure (like graphs). This information must be provided by other means.

Control-Flow Automaton

CPAchecker automatically computes a control-flow automaton (CFA) of the verified program. This CFA can be saved in a file to support understanding of the verification results or simply to examine the control-flow structure of the program. The information is stored in DOT-format[2]. DOT is the input format for the graph visualization software Graphviz[3].

The DOT format is human-readable because it is encoded in the UTF-8 character encoding rather than in binary encoding. But even small example programs

[2] Definition of DOT-format at `http://www.graphviz.org/doc/info/lang.html`
[3] `http://www.graphviz.org/`

generate graphs that cannot be easily interpreted by humans in DOT representation. For this reason output information in DOT-format should be presented as graph in a graphical user interface context. In this case the DOT file could contain meta information for vertices and edges like links to the corresponding part in the source file. This information would be used in the graphical representation to enhance the usability.

Abstract Reachability Tree

The abstract reachability tree (ART) is a graph that shows all program execution paths that were checked by the verification tool. If the CPAchecker analysis has terminated with *No_Error_Found* result the ART is regarded as a proof of this result. The ART is saved in DOT-format like the CFA and similar considerations apply to its presentation.

Error Path

An error path is a program execution path that results in an error as defined in the specification. If CPAchecker finds an error during the verification process it generates the program execution path that lead to this error. This error path can be saved in a text format. It is important that a user can easily understand an occurred error path because it represents either an error in the program or an error in the specification. Either way it should be easy for the user to decide which document is not defined correctly and why the error occurs.

There are several possibilities to support the user in this task. The first and most simple possibility is to provide a link from every node in the error path to the corresponding source file. This helps the user to discover the context of the error. A more sophisticated possibility is to add the error path information to the ART. As further enhancement the error path nodes in the ART could be extended with additional meta information. This meta information could be the value of variables in the program state or predicates that are fulfilled in the state. In a graphical representation of the ART these information can for example be shown in a tool tip.

The configuration option "analysis.stopAfterError" can be used to find more than one error. If the default configuration value is used the analysis stops when the first error path is found but the program might contain more errors. To find these further errors the analysis has to be run again when the error is fixed. If the user configures the option to find all reachable errors at once the analysis might find more than one error path. To display more than one error paths in the ART will probably confuse the user. Instead of displaying multiple error paths at once the user interface could provide a way to switch between error paths. This would result in a user interface that is always centered on one error and allows the user to concentrate on this single error without being influenced by other annotations. This way of cycling through error paths can only be effective if the number of reachable errors is small and the verification tool generates only few false positive results. If the tool should support the handling of a bigger number of error paths at once this would require an error management interface that shows the user which error is displayed and which other errors are available.

To reproduce the found error path and test the program after the error has been fixed the user needs a test configuration. This configuration contains the program parameters for executing it along a given error path. This test configuration would allow ensuring that this particular error path has been fixed without running the (possibly expensive) verification process again. In principle this test configuration can be generated from an error path by following the path from end to beginning. While following the path all program variables would be monitored and set so that the program execution path represented by the error path can be recreated. When the beginning of the error path is reached this method will result in a set of variables. Some of these variables might not have values attached because the variable was not used on the error path. To get a test configuration all program parameters have to be assigned the value of the corresponding variable. If no value is assigned to the variable a default value may be assigned.

Generating this test configuration from the text based error path would be complicated because the code at each node along the path has to be parsed and interpreted. It would be more appropriate to use the error path representation held by the verification tool in memory. For the tool BLAST this is described in [BCH+04a].

3.6 Reusable information

Because the verification process can take a significant amount of time, the verification tool should provide means to reuse as much work as feasible. Possible approaches have been described in the background Chapter 2.4. As CPAchecker has a modular architecture it must be left to the individual analysis to decide what information can be reused and what must be recomputed. The decision whether an information that is computed should be stored for reuse in a later verification run should be made considering the following aspects:

1. time invested to compute the information,
2. time that needs to be invested to save and restore the information, and
3. how likely is it that the information can be reused if the source file or the specification is slightly changed

One way of allowing analyses in the modular CPAchecker architecture to reuse information is to provide a verification task specific output directory where the analyses can save computed information. When the verification task is run again this directory can be provided as input directory so the analyses can reload the saved information files. This method allows a flexible implementation of the reusable information feature. The developers of analyses that use this feature can decide independently if they reuse parts of the saved information and how they save the information in files. Because these files are designated for reuse by the verification tool and not by the user they do not need to be encoded in a text based file format. It might be more efficient to use a binary encoding in this case. The configuration and specification that was used to compute the information must be saved with the reusable information. This way future analyses that want to reuse this information can determine under which configuration it was computed. If the configuration was changed in the meantime, parts of the information might not be reusable.

3.7 Verification-Task Collection

In this section we introduce the concept of verification task collections. A collection is a simple set of verification tasks and can be the element of another collection. This results in a tree-structure where the inner nodes are collections and the leaves are verification tasks.

These collections can be used to add additional structure to the verification process. The tasks of one verification project can be separated by an aspect like which source code module is tested or which type of specification is verified. Concrete aspects to separate verification tasks are project specific and perhaps even dependent on structure of documents like the functional specification document which is drafted in the requirements engineering.

The additional structure of verification task collections can be used to improve the user interface. A complete task collection can be verified with one user interaction and run without further interaction. Because there are no dependencies between verification tasks they could be run in different threads in parallel. This would improve the overall runtime of verification task collections. Users should also be able to verify a selected subset of a task collection. This way the user could avoid running verifications that are not interesting at the moment but would consume considerable amounts of time.

3.8 Verification Tasks in CPAchecker

This section describes how verification tasks are integrated into the CPA framework [BHT07]. The implementation of verification tasks in CPAchecker and in the Eclipse-Plugin of CPAchecker corresponds to this description. We take for granted that the specification is translated into a set of CPAs. This will be explained in Chapter 4.

At the beginning of a verification run CPAchecker generates a tree of configurable program analyses from the configuration file. The inner nodes of this tree are analyses that wrap other analyses and process the intermediate results of the wrapped CPAs.

When the specification is parsed it generates a set of CPAs. Each element of this set represents one automaton as defined in the specification. These CPAs are then injected into the wrapper CPA[4] at the root of the tree of CPAs. If the

[4] A CPA that can contain other CPAs

tree root is no wrapper CPA an error is issued and the analysis stops. Another possibility to handle this issue would be to replace the root of the tree with a new wrapper CPA that wraps the previous root and all CPAs generated from the specification. However having no wrapper CPA as root is an unusual case because some simple analyses[5] are required in almost any use case. If one of the simple analyses is used with a more complex analysis the root of the CPA tree has to be a wrapper CPA. The specification CPAs are then injected into this wrapper CPA.

Because the specification is implemented as a set of CPAs there is no need for further special treatment of the specification. For the rest of the verification process the specification CPAs can be handled just like any other CPA.

[5] like the LocationCPA or the TypesCPA

Chapter 4

Specification Language

4.1 Overview

The specification language for CPAchecker was developed in this thesis. CPAchecker had already a language for specification of automata[1] that was extended and included in the specification language.

The CPAchecker specification language is defined by the "Specification Language Grammar" in Appendix 7.1. The top level elements of the grammar are include statements and specification-statements. "Include" statements are used to include other specification files in a syntax that is similar to the `#include` preprocessing directive of c. Further discussion of include statements is given in Section 4.2.

Specification statements are used to define which program paths must be considered errors. In the CPAchecker specification language automata are used to define these program paths. To fulfill the Requirement 1 as defined in Section 3.3, the language provides an easier way to write certain simple specifications. This abbreviated automata notations are described in Section 4.4. The discussion of the automata definition language is in an extra Section 4.3.

The CPAchecker specification language is designed to be an extensible language for research purposes. When the need for new language constructs becomes apparent these constructs can be easily integrated. At this point the specification language allows the three discussed top-level statements. It could be extended by adding more top-level statements that would allow easier expression of certain specifications. For example relational specifications like in BLAST [BCH$^+$04b] could be added here.

Other verification tools like SLAM [BR02] allow the user to specify properties using C code. This is problematic for CPAchecker because the specification is interpreted by the tool. SLAM and BLAST use source code annotation to verify the specification. So the specification parts that allow C-statements can be copied into the source file and are handled by the C-compiler. CPAchecker does not have this option because source code and specification are kept separate.

Allowing C as language for parts of the specification has the advantage that the users (which are probably C programmers) can easily write the specification without learning new syntax and semantics. On the other side we see the disadvantage that code might be copied from the source program code into the

[1] The original automaton language was developed in a term paper.

specification. If this happens, errors in the source code might be copied into the specification. The specification would be faulty from the very beginning on. The verification would never find the error in the source code because according to the (faulty) specification the source code is correct. Experiments and user feedback have to show if this theoretic consideration holds in the real world.

Following this argumentation the CPAchecker specification language has an expression definition language that is similar to a simplified C language. To keep the language as simple as possible we did not include control flow constructs like if-branches or loops. As the specification language is similar to existing programming languages it should be easy to learn for program developers.

4.2 Include

Include statements allow the inclusion of other specification files. To provide a convenient user interface, the syntax of C *include* directives was reused for the CPAchecker specification language. As CPAchecker is a tool for verification of C programs the syntax should be intuitive for users.

```
#include sub_specification.spc
```

Listing 4.1: Example: Include

With this statement the user is able to use pre-defined specifications and to avoid duplication of specification code. Therefore the language fulfills the Requirement 2 (defined in Section 3.3) for specifications. One problem with this include statement is, that the user is allowed to write cyclic inclusion references like in the example below. In this example the specification file “cyclic1.spc” contains an include statement that references the same file. This creates a very simple cycle because “cyclic1.spc” references itself.

```
#include cyclic1.spc
```

Listing 4.2: cyclic1.spc

If the specification would be more complex and the cycle would contain more than one specification file it might be hard to detect for the user. The specification language parser avoids cycles by detecting multiple references to specification files. An example for a problematic specification file with include statements is given in Figure 4.1. In the example a specification file A includes the specification files B and D. D includes B and E and the file E includes D again. This example contains a multiple reference to B and a cycle consisting of the files D and E.

CPAchecker avoids multiple references (which includes cycles because D is referenced twice) by evaluating only the first reference to a file. Every further reference generates a warning message and is not evaluated. Generally it might be useful for a user to reference files multiple times, but in the current specification language multiple references can be ignored. This is because the specification language contains only elements that cannot reasonably be declared more than once. Automata (which are so far the only other elements of the language) can

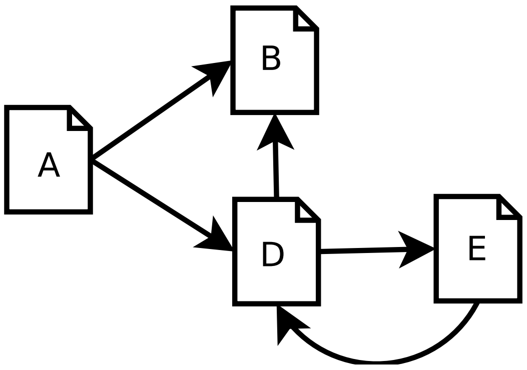


Figure 4.1: Example for a Cyclic Specification

in theory be run in parallel but if they are defined identically they would always take the same actions and always be in the same state. So it does not make sense to declare automata twice and therefore it does not make sense to include completely identical files more than once. When the specification language is extended with more elements it must be determined whether this condition still holds.

4.3 Automata

4.3.1 Automata as CPA

The automata that are defined in the specification language are integrated as a CPA+ into CPAchecker. This subsection defines how automata are integrated into the formal CPA framework [BHT08] as described in Subsection 2.2.2.

The *AutomatonCPA* $\mathbb{A} = (D_\mathbb{A}, \Pi_\mathbb{A}, \rightsquigarrow_\mathbb{A}, merge_\mathbb{A}, stop_\mathbb{A}, prec_\mathbb{A})$ consists of an abstract domain $D_\mathbb{A}$, the set of precisions $\Pi_\mathbb{A}$, a transfer relation $\rightsquigarrow_\mathbb{A}$, a merge operator $merge_\mathbb{A}$, a termination check $stop_\mathbb{A}$ and the precision adjustment function $prec_\mathbb{A}$. The automaton[2] for this CPA is $A = (S, s_0, I, T)$ with the set S of automaton states, the initial automaton state $s_0 \in S'$, the set of integer variables I and the set of transitions T. Each of the automaton components is described in the following section. The components of $\mathbb{A}$ are:

1. The abstract domain $D_\mathbb{A} = (C, \varepsilon, [\![\cdot]\!])$ where $\varepsilon = (E_\mathbb{A}, \top, \bot, \sqsubseteq, \sqcup)$ is a semi-lattice.
   The abstract states of the domain are $E_\mathbb{A} : (V \to R, \{OK, incomplete\})$ with $V = (state, i_0, \ldots, i_{|I|})$ and $R = S \times \mathbb{N}^{|I|}$. The set V consists of a variable 'state' for the current observer state and an integer variable for each observer variable. E assigns an observer state to the 'state' variable and an integer value to each of the integer variables. The second component of the abstract state contains information on the processing of the state by the

[2] As introduced in Section 2.3.

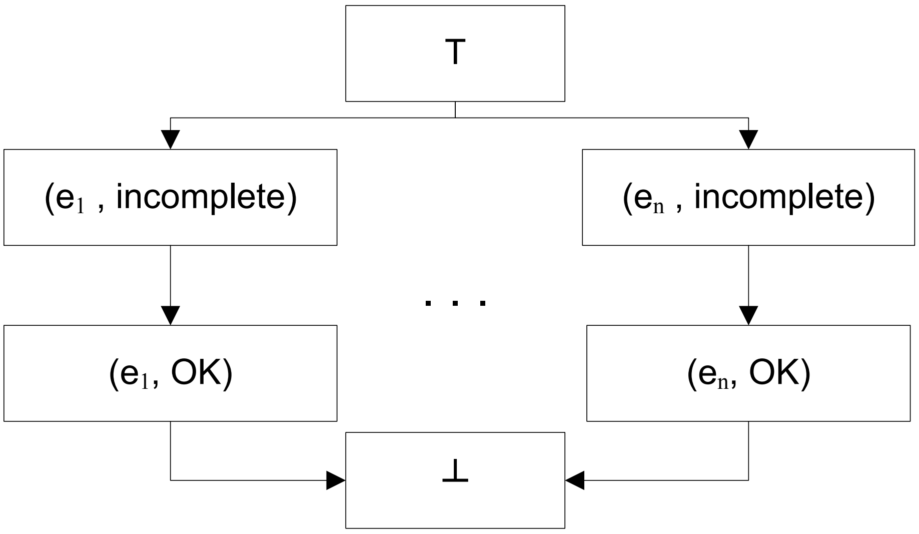


Figure 4.2: Lattice for AutomatonCPA

verification algorithm. The value *incomplete* means that more information is needed to compute the successor state (this computation can be done by the strengthen operator $\downarrow$). It means that the following abstract states can only be computed with additional information from other CPAs. This information will be processed by the *strengthening* operator $\downarrow$.
The top and bottom elements of the lattice are denoted by $\top$ and $\perp$, respectively.
A tuple $((s,p),(s',p') \in E_{\mathbb{A}}^2)$ is element of the relation $\sqsubseteq$ if $((s',p') = \perp$ or $(s',p') = \top$ or $(e.state = e'.state \wedge (p = p' \vee p = OK)))$. The join function $\sqcup$ is defined as $\sqcup = \{(e,e') \in E_{\mathbb{A}}^2 | e' = \top \vee e = e'\}$. The concretization function assigns a subset of the set of concrete states C to each abstract state. The semi-lattice ε is illustrated in Figure 4.2.

2. The set of precisions $\Pi_{\mathbb{A}} = \{default\}$ contains only a default precision. The AutomatonCPA does never alter its precision and uses the precision adjustment feature only to signal the termination of the analysis with the precision adjustment operator $prec_{\mathbb{A}}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{A}} \subseteq E_{\mathbb{A}} \times G \times E_{\mathbb{A}}$ where G is the set of transitions of the CFG. A tuple $((s,p),(s',p'))$ is in $\rightsquigarrow_{\mathbb{A}}$ iff all transitions of s can be evaluated and s' is the resulting state of one of these transitions or if at least one transition of s could not be evaluated and $p' = incomplete$. This result can be resolved by the $\downarrow$ operator which has additional information.

4. The operator $merge_{\mathbb{A}} : E_{\mathbb{A}}^2 \to E_{\mathbb{A}}$ is defined as $merge_{\mathbb{A}}(e,e') \to e'$ and the operator $stop_{\mathbb{A}} : E_{\mathbb{A}} \to \mathbb{B}$ is defined as $stop_{\mathbb{A}}(e) \to (\exists e' \in R : e \sqsubseteq e')$.

5. The precision adjustment function $prec_{\mathbb{A}} : E_{\mathbb{A}} \times \Pi_{\mathbb{A}} \times 2^{E \times \Pi} \to E_{\mathbb{A}} \times \Pi_{\mathbb{A}} \times B$ where $B = \{break, continue\}$ (see Section 2.2.2) determines if the automa-

ton has reached a state where the analysis should terminate (the automaton has detected an error). The function is defined by $prec_{\mathbb{A}}(e, \pi, reached) \rightarrow (e, \pi, b)$ where $b = break$ if e is an error state and $b = continue$ otherwise.

The enclosing CPA should use a strengthening operator as follows: $\downarrow$: $E_{\mathbb{A}} \times \mathcal{P}(E) \rightarrow E_{\mathbb{A}}$. The operator strengthens the abstract state (s, p) by using additional information in form of abstract states of the other used CPAs. The operator meets the requirement $(p = OK) \Rightarrow (e, p) = (e', p')$. This strengthening operator is used by the composite analysis [BHT07].

4.3.2 A Simple Example Automaton

This section illustrates the use of automatons on a simple example. This example contains the three observer states `Init`, `Locked` and `Unlocked`. The transitions between these states are triggered by occurrences of “init()” and “unlock()” on transitions of the CFA. This example automaton expects to see an “init()” and alternating “lock()” and “unlock()”. If any of these expressions occur out of order the automaton reaches an error.

```
OBSERVER AUTOMATON LockingAutomaton
LOCAL int counter = 0;
INITIAL STATE Init;
STATE Init :
  MATCH  "init()"   -> GOTO Unlocked;
  MATCH  "lock()"   -> ERROR;
  MATCH  "unlock()" -> ERROR;
STATE Locked :
  MATCH  "init()"   -> ERROR;
  MATCH  "lock()"   -> ERROR;
  MATCH  "unlock()" ->
    ASSERT counter != 0 GOTO Unlocked;
STATE Unlocked :
  MATCH  "init()"   -> ERROR;
  MATCH  "lock()"   ->
    DO counter = counter+1 GOTO Locked;
  MATCH  "unlock()" -> ERROR;
END AUTOMATON
```

Listing 4.3: Definition of the Locking Automaton

```
int locked;
int lock() {...}
int unlock() {...}
int init() {...}

int main() {
  init();
  lock();
  unlock();
  return 0;
}
```

Listing 4.4: Correct C-Program for the locking automaton

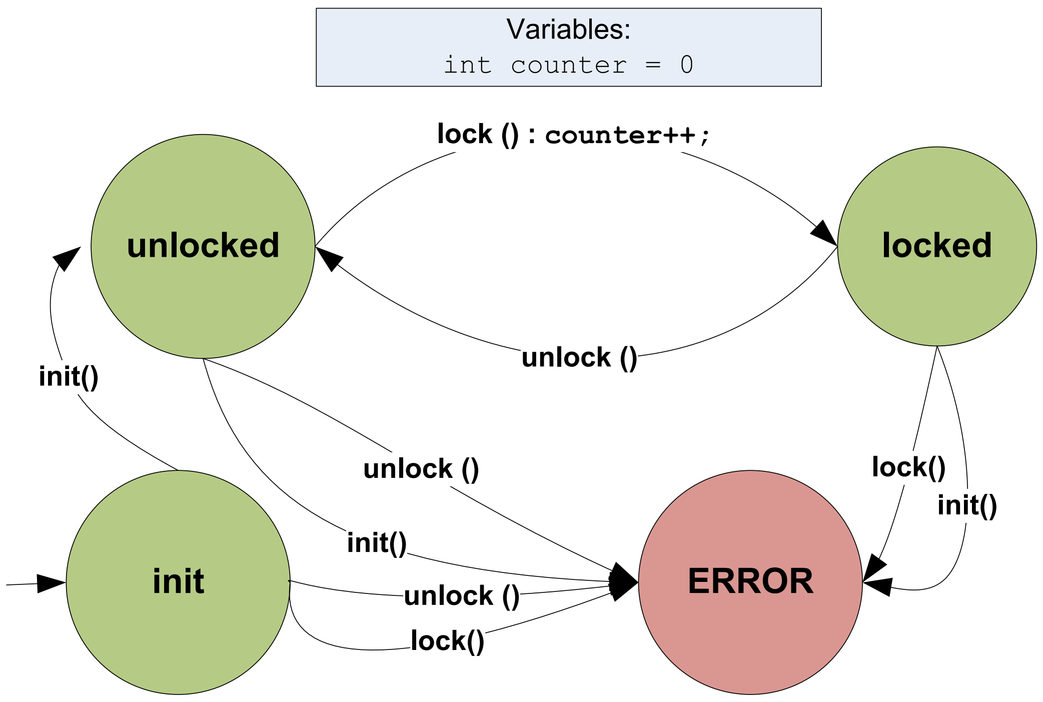


Figure 4.3: Locking Automaton as Graph

This program is considered correct because the locking automaton does not reach the error during analysis of the program. To further illustrate the example automaton we present it in a graph layout in Figure 4.3.

4.3.3 Automata Architecture

The components of an automaton are described in the following:

1. The name of the automaton. This should be a globally unique identifier because only one automaton per identifier is permitted.
2. The type of the automaton. This can be either **`OBSERVER`** or **`CONTROL`**. Observer-Automata are not permitted to perform certain actions like modifying other CPAs.
3. Integer variables that can be used to store state-independent information.
4. States that make up the finite part of the AutomatonCPA states (in theory the variables are infinite).
5. Transitions that control the change of the AutomatonCPA states.

The complete grammar for the automaton language is given in the appendix (Section 7.1). The discrimination of observer and control automata gives the user additional control on what the used automata are capable of. The verification result is always heavily dependent on which automata are used, because the automata define what is considered an error in the program. But if the user includes control automata, that might modify or disable other analyses. If analyses are disabled some errors might not be detected even if they are specified in **OBSERVER** automata. Therefore such possibly problematic automata have to be marked with the keyword **CONTROL**.

Concretely, in **OBSERVER** automata the commands **MODIFY** and **STOP** must not be used because they manipulate the behavior of the other CPAs. These keywords are both described in Section 4.3.6. The listing 4.5 presents the general layout of automata. Keywords are printed bold. Further details will be given in the respective sections.

```
OBSERVER AUTOMATON name
LOCAL int var_name_1;
INITIAL STATE state_name_1;
STATE state_name_1 :
  matching_statements -> actions;
  matching_statements -> actions;
STATE state_name_2 :
  matching_statements -> actions;
  matching_statements -> actions;
END AUTOMATON
```

Listing 4.5: General layout of automata

The question whether the declared automaton is finite (the automaton has only a finite number of states) or infinite requires some more detailed discussion. As the user can only write a finite number of **STATE** declarations their number is definitely limited. This automaton has the set of states S_1 with one element per **STATE** declaration and is finite. However when we take into consideration the local variables $I_1, ...I_n \in \mathbb{Z}$(which are theoretically infinite) a new automaton emerges. This new automaton has a set of states $S_2 \subseteq (S_1 \times \mathbb{Z}^n)$. The elements of S_2 consist of an automaton state (as declared with **STATE**) and a value for each automaton variable. This means even when the automaton declares only one variable the state space is infinite. Of course CPAchecker simulates this second automaton. The first, finite automaton can be saved as a DOT-file for debugging or documentation purposes.

4.3.4 States, Variables and Transitions

The main components of an automaton are states and transitions. An automaton consists of a set of states S and a set of transitions T. One of the states is identified as the initial state. The initial abstract state of the AutomatonCPA consists of this initial state and initial values for all automaton variables. Automaton variables are declared with the keyword **LOCAL**. They can be assigned an initial value in standard C-Syntax. If this assignment is omitted, the initial value is 0.

The transitions consist of a matching function and actions. These components are separated by the keyword `->`.

When the verification algorithm processes a CFA edge the AutomatonCPA analyzes this edge and determines whether any of the transitions, which are starting in the current state, fire. If any of the transitions fire (if the *matching function* of the transition evaluate to true), the AutomatonCPA executes the actions on this edge. One of the actions will always be either the declaration that an error is reached (keyword **`ERROR`**), the termination of the analysis path (keyword **`STOP`**) or the selection of the next automaton state (keyword **`GOTO`**).

Exchangeability of Variables and States

The AutomatonCPA stores the state of the analysis by remembering the automaton state (a state defined with the **`STATE`**) and the values of the automaton variables. Normally a user would use both constructs in combination, because each has strengths and weaknesses. However it is possible to replace almost all states by encoding in variables and almost all variables by encoding them into states. One limitation is that at least one state is needed as initial automaton state. Every other state can be encoded as value of an additional integer variable. This would result in a single-state automaton with very many transitions that would mainly test which value the state-variable currently has. This kind of automaton would be cluttered because of the loss of structure.

The other extreme is using no variables and encoding every possible combination of variable values as states. Even in theory this is impossible because the integer variables can have infinite values and we can not encode these values in a finite set of automaton states. So the resulting automaton would not be a finite automaton. In our implementation the automaton variables are Java [GJSB05] integer variables that have a finite state space. So this approach would be feasible in practice. The resulting automaton would contain very many states and the same transitions would have to be duplicated in different states.

To optimize usability we let the user decide how many states and how many variables should be used for concrete problems.

Transition Evaluation Strategies

The CPAchecker specification language allows two strategies for firing of transitions.

1. **`USE_ALL`** All transitions of the current state are evaluated. If the matching functions of multiple transitions evaluate to `true` a new abstract state for each of these transitions is generated. If no transition matches no abstract state is generated and the analysis path terminates.

2. **`USE_FIRST`** The transitions are evaluated in the sequential order in which they were declared in the specification. If one transition matches its actions are executed and the following abstract state is generated. The other

transitions are not evaluated. If no transition matches the AutomatonCPA stays in the current state.

Which strategy should be used can be declared for each state separately by using the keywords **`USE_ALL`** or **`USE_FIRST`**. If no strategy is declared **`USE_ALL`** is used.

4.3.5 Functions

This section describes the functions of the automaton definition language that do return values.

During the evaluation of the AutomatonCPA the automaton is in a certain automaton state s. When the CFA takes a transition the AutomatonCPA has to decide, if this transition in the CFA triggers a transition in the automaton. This decision is taken by evaluating the matching functions of the transitions of s.

Matching functions are boolean functions. If the function evaluates to `true`, the transition is taken as described in the paragraph “Transition Evaluation Strategies” in Section 4.3.4.

The boolean functions described in this section can generally also be used in the “action” part of the transition definition. The matching functions *exact matching*, *AST comparison matching* and *regular expression matching* are meant primarily to be used in the matching function part of the transition declaration. They can also be used as boolean functions in the action part of the transition. To provide a better layout for the reader of this document we will make some notational statements here. We will use the types boolean and integer in the following sections. They will appear in bold typesetting (**bool** and **int**) in the text.

Basic Operations

The CPAchecker implements the basic operation of the boolean algebra and basic arithmetic operators. Constant boolean values are **`TRUE`** and **`FALSE`**. Constant numeric values can be used by simply writing them in the specification document. The automaton variables can be used by writing their identifier in any place where an **int** type can be used. The binary operators **`AND`** and **`OR`** and the unary operator `!` (which is the negation) can be used to combine other boolean functions. The parameter type and the result type of these functions is **bool**. For algebraic operations we only implement the binary operators `+` and `-` with argument and result type **int**. All binary operators described in this paragraph are written in infix notation.

Exact Matching

bool **`MATCH`** `"pattern"`
The string *pattern* following the MATCH keyword is considered an exact match iff it is exactly the same as the sourcecode associated with the CFA edge. The automaton in Section 4.3.2 contains examples of this matching method. If the pattern is matched exactly the function returns **true**, otherwise it returns **false**.

AST Comparison Matching

bool **`MATCH`** `{pattern}`
For an AST comparison matching the automaton generates an abstract syntax tree (AST) from the string *pattern* between the curly brackets in the definition. The AST-generation is delegated to the CDT C/C++ IDE for ECLIPSE that is already used in the CPACHECKER framework. The generated AST is compared to the AST associated with the CFA transition and the comparison result is returned by the function. To allow more flexibility in defining transitions the comparison allows wildcards and capturing of parts of the source code. For every occurrence of `$?` any identifier or literal in the sourcecode (which is situated in the same position in the AST) will be considered equal. If `$1` (or any other number instead of 1) is used the identifier or literal is not ignored but captured for later reference by other functions in the same transition. The following examples illustrate the use of an AST comparison transition:

- The pattern `x = 5;` matches on `x=5;` and `x= 5;` but does not match on `x=10;` or `y=5;`.
- The pattern `$? = 5;` matches on `x=5;` and `y=5;` but does not match on `x=10;`.
- The pattern `$? = $?;` matches on `x=5;`, `x=y;`, `x=funct();`.
- The pattern `$1 = $2;` matches on the same statements as the example above. As it matches on `x=5;` the *transition variable* `$2` will contain the string “5”. If a transition variable is used in a position where an **int** is expected it will be converted to an **int**. If the conversion fails the user is informed.
- The pattern `init(argA, $?);` matches on `init(argA, xy);` but not on `init(x,y);`.

Regular Expression Matching

bool **`MATCH`** `[pattern]`
For even more flexibility the regular expression matching function allows to match on any Java regular expression [3]. The use of this method results in relatively ugly code because many C-tokens like “(“ must be escaped. The result of the Java regex matching is returned as result of the function.

Label Matching

bool **`MATCH LABEL`** `[pattern]`
This function can be used to match on a label in the C-source code. Iff the current CFA-edge is annotated with a label L and L is matched by the regular expression *pattern*, this function returns `true`. The use of regular expressions is

[3] The syntax of Java regular expression is defined in the Javadoc of the Java Pattern class.

not problematic in this context because label identifiers normally do not contain characters that would have to be escaped in regular expressions.

Matching of the Program Exit

bool `MATCH EXIT`
This function determines if the currently processed part of the CFA is terminating the program. In more formal terms the function tests whether the node that the current CFA edge leads to has any outgoing edges. If the node has no outgoing edges the current edge is the last edge of a program path. This function can be used to express that the automaton must not be in a certain state when the program terminates. For example it is used in the Locking automaton in Section 4.5.1. In that example it is used to ensure that the lock is not engaged (that the automaton state is not "Locked") when the program terminates.

CPA querying

bool `CHECK(cpa_name, "Query-String")`
The CPA querying function allows to query any other CPA that is evaluated in the same CompositeCPA as the AutomatonCPA. When the function is evaluated it searches for a state of the identified CPA and sends the query string. The answer (which must be of **bool** type) is returned as function result. If the AST comparison matching (Section 4.3.5) is used and a part of the match was captured as \$1 in the same matching function as \$1 this part will replace \$1 in the query-string. Each other CPA can define an own syntax that is accepted as query string. An example querying of the CPA "Explicit CPA" is given in the following listing:

```
CHECK(ExplicitAnalysis, "main::st==1")
```

Listing 4.6: Example for CPA querying

The example queries the "ExplicitAnalysis" whether the variable "st" in function "main" has the value 1. Because the function requires additional information (the abstract element of the other CPA) it cannot be evaluated in the transfer relation and the evaluation must be carried out in the strengthening operator (described in Subsection 4.3.1).

var `EVAL(cpa_name, "Query-String")`
The EVAL function can be used just like the CHECK function. In contrast to the CHECK function EVAL allows an arbitrary return value. The return value is cast depending on which other function or action uses the result of the EVAL statement. It can for example be used to query other CPAs for string messages and print them to the log with a PRINT statement.

4.3.6 Actions

This section describes statements without return value in the automaton specification language. These statements are used as top-level statements in the *actions* part of the automaton specification. The actions part consists of a list of action statements.

Print Statement

`PRINT <arguments>`
The print statement prints a message to the user on the log. The `<arguments>` are a list which elements can be strings (surrounded by quotes), **int** functions or **bool** functions. The elements of the list will be evaluated in the order in which they occur in the list and the results will be printed on the log. There are some special commands that can be used inside string elements:

- *$rawstatement* The string $rawstatement is replaced by the source code string that is associated with the current CFA edge.
- *$line* The string $line is replaced by the number of the source code line associated with the current CFA edge.
- *$1* If the AST comparison matching (see Section 4.3.5) is used and a part of the match was captured as $1 this part will replace $1 in the message.
- *$$varname* This command will be replaced with the current value of the automaton variable "varname".

Assignment

`variable = value`
The assignment statement assigns the value of an **int** function to the identified automaton variable.

Modification Statement

`MODIFY (cpa_name, "Modification-String")`
This **MODIFY** statement is similar to the **CHECK** function described in Section 4.3.5. The difference is that the CPAs accept other strings if **MODIFY** is used and **MODIFY** does not have a return value. **MODIFY** is used to change internal information in the abstract state of other CPAs. In contrast to this, **CHECK** is used to query abstract states of other CPAs and get a boolean result. Because the command allows the modification of other analyses it is only permitted in control automata. An example for a modification is given in the following listing.

```
MODIFY(ExplicitAnalysis, "deletevalues($2);setvalue($2:=0)")
```

Listing 4.7: Example for CPA modification

In this listing a program variable name was captured in an AST comparison matching in $2. Then all values for the program variable are deleted from the abstract state and 0 is set as new value. In the end the abstract state for the explicit analysis has only one value (0) for the variable.

Assertion Statement

`ASSERT` `condition`
The assertion statement is used to ensure that a given condition is true. If the statement is run it evaluates the given condition (which must be of type **bool**). If the result of this evaluation is true the statement does nothing more. If the result is false the automaton declares that an error is reached. The statement **`ASSERT FALSE`** is equal to the statement **`ERROR`**.

Goto Statement

`GOTO` `nextState`
This statement declares which state the automaton should be in after the transition is taken (and no **`ASSERT`** statements failed). The abstract state that is generated from this transition has "nextState" as automaton state. The argument "nextState" must be the name of a state declared in the same automaton.

Stop

`STOP`
The stop statement declares that this transition generates no following abstract state. If no other transition generates an abstract state the analysis for this program path is terminated without reaching an error. The other analyses that are investigating the same program path are also terminated. Therefore this command heavily influences other analyses (without declaring an error) and is only allowed in control automata.

Error

`ERROR`
The error statement declares that the automaton has identified the program path leading to the current state as an error. If this happens, the analysis terminates[4] with *Error_Found* as primary verification result (see Section 3.5).

4.4 Abbreviated Automaton Notation

As stated in Section 3.3 simple requirements should be simple to be declared in a specification. The automaton definition language described in the previous section allows the declaration of complex automata. So there is still the need for a simplified statement to be used for simple specifications. One example for such a statement is that the verification tool should test, if the source program uses the function `free()` on a pointer that is potentially already freed. CPAchecker comes with a CPA that already detects this situation (the PointerAnalysisCPA). But the PointerAnalysisCPA does not report this situation in the default configuration because this program behavior might be the user's intention. The fact that

[4] The analysis terminates only if the current program path is found to be feasible.

the situation occurred can be queried with a **CHECK** statement. So the statement **ASSERT** ! **CHECK**(PointerAnalysis, "DOUBLE_FREE") would do the right thing if it was embodied in a transition in an automaton. Unfortunately the work required for defining the automaton would be quite much in relation to writing this assertion statement.

This is why the CPAchecker specification language allows writing single assertion statements as third [5] top-level statement. The assertion statement is embodied in a skeleton automaton during the parse process. In the following listings an example assertion and the finished automaton are shown.

```
ASSERT ! CHECK(PointerAnalysis, "DOUBLE_FREE")
```

Listing 4.8: Example for a Simple Assertion Statement

```
CONTROL AUTOMATON AnonymousAutomaton1
INITIAL STATE OK;
STATE OK :
  TRUE -> ´
   ASSERT ! CHECK(PointerAnalysis, "DOUBLE_FREE")
   GOTO OK;
END AUTOMATON
```

Listing 4.9: Skeleton Automaton with the Simple Statement Inserted

4.5 Specification Examples

This section presents examples specifications with source code programs. A step-by step description of the verification process is given to further illustrate the usage of the specification language.

We describe the two principal methods for notation of specifications for software verification. In the *source code annotation* approach the specification is written in the source code file. This might be as distinct annotations in non-code parts like comments (an example for this are Java annotations) or as statements in the source code language. In our approach the specification is written in an extra file to support the separation of concerns. This section will compare the two approaches using a concrete example.

4.5.1 Locking Example

This example is similar to the simple locking example given in Section 4.3.2. In fact this example uses a simplified version of the automaton from the previous example. The only addition is the last transition in state “Locked”, which will be explained in this section. In contrast to Section 4.3.2 this section focuses on the description of the verification process.

The last transition of the automaton state “Locked” ensures that the analysis finds an error if the program terminates while the automaton is in the state

[5] The other top-level statements are include statement and automaton definitions.

```
OBSERVER AUTOMATON LockingAutomaton
INITIAL STATE Init;
STATE Init :
  MATCH "init()"-> GOTO Unlocked;
  MATCH "lock()" OR MATCH "unlock()" -> ERROR;
STATE Locked :
  MATCH "unlock()" -> GOTO Unlocked;
  MATCH "init()" OR MATCH "lock()" -> ERROR;
  MATCH EXIT AND (! MATCH "unlock()") -> ERROR;
STATE Unlocked :
  MATCH "lock()" -> GOTO Locked;
  MATCH "init()" OR MATCH "unlock()" -> ERROR;
END AUTOMATON
```

```
 1 if (a) {
 2   init();
 3   lock();
 4   // ...
 5   unlock();
 6 }
 7 if (b || c) {
 8   lock();
 9   // ...
10   unlock();
11 }
```

Source Code for the Locking Example

LockingAutomaton

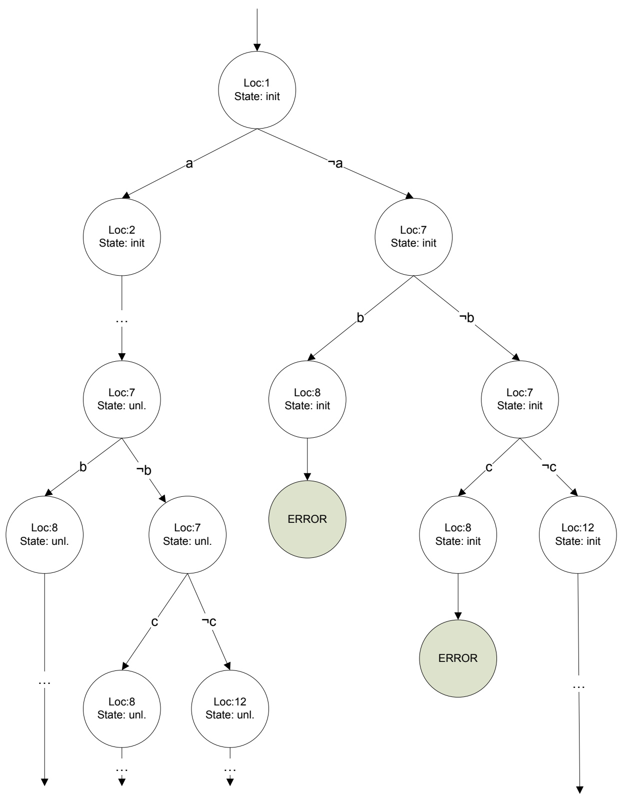


Diagram for the Locking Example

Figure 4.4: Locking Example

"Locked". Because the functions **MATCH EXIT** and **MATCH** "unlock()" could be true on the same edge this case has to be considered. If the last statement of the program is `unlock();` the analysis should not find an error, so this case is excluded from the **MATCH EXIT** transition.

Figure 4.4 shows the source code fragment of the locking example and a diagram illustrating the verification process. The diagram is given only informally, some nodes are omitted. Each node is annotated with the corresponding state information computed by the verification algorithm. The diagram is almost an AST for the source code fragment. The difference is that in an AST nodes with equal information would be identical. That means for example the two nodes with information "loc:8 State:init" would be identical and there would only be one circle to represent this node.

The annotation of each node has two values. The property "loc" identifies the number of the source code line that is currently processed. The property "State" marks which state the locking automaton is currently in. The two nodes marked with "ERROR" state that the automaton has detected an error and the verification terminates.

At the beginning of the verification of the code fragment the current state consists of the program location 1 (which stands for source code line 1) and the automaton state "init". The first code statement is a conditional branch with condition "a". Therefore the initial node in the diagram has two successor states. Depending on the value of "a" one of these states must be the next state. Because the variable is not initialized the verification algorithm has to explore both possible paths. First the algorithm assumes that the value of "a" is *true* and therefore the left successor node is generated. Later the algorithm will return here, assume that "a" is false and generate the right successor node. The complete tree is explored in this fashion until one of the "ERROR" nodes is reached. When this happens the algorithm terminates and gives the path to this error to the user.

The specification of this example has been used to identify two paths in the control flow automaton as errors. If source code annotation was used to accomplish this, it would have been necessary to add a *assert* statement before every call of the library functions.

There are two basic parts of code when a program uses a library, the *library source code* itself and the *calling code* which is using the library. In this example only the calling code is given. The library source code is not necessary to verify the calling code. To write the specification the user only needs to know the preconditions assumed by the library functions. This independency on the source code is helpful, because when using the proprietary libraries the source code is normally not available.

4.5.2 Clock Example

This example shows a small function for setting the hour value of a clock (this is an example for a library). The function takes a parameter `p_hour` and assigns its value to the private variable hour. To work correctly the function assumes that the value of `p_hour` is between 0 and 23. This is the specification that has to be verified.

```
OBSERVER AUTOMATON clock_automaton;

INITIAL STATE init;

STATE init :
  MATCH{hour = $1} ->
    ASSERT (%1 < 24)
    ASSERT (%1 > -1)
    GOTO init;

END AUTOMATON
```

Specification for the Clock Example

```
int hour = 0;

int set_hour(int p_hour) {
  hour = p_hour;
}
```

Source code without annotation

Figure 4.5: Clock Example with separate Specification

```
int hour = 0;

int set_hour(int p_hour) {
  if (p_hour > 23 || p_hour < 0) {
    printError("Internal Program error");
  } else {
    hour = p_hour;
  }
}
```

Source code without annotation

Figure 4.6: Clock Example with in-code Specification

This specification implies certain behavior of the *calling code* (the calling code has to provide parameters in the correct range). Therefore a verification of this specification would verify the calling code rather than the library code. The specification can only be violated by the calling code. But the specification can be written without knowledge of the calling code. If this library would be proprietary the developer could write the specification and provide it with the binary release of the library. Then it could be verified with the calling code without having the library source code. This is equivalent to the situation of the example in Section 4.5.1.

This example shows a version of the specification written in source code (Figure 4.6) and a version using our specification language (Figure 4.5). As we already emphasized the source code annotation approach does not support the separation of the concerns of specification and code. Therefore it does not enable the delivery of a formal specification with the binary release of the library as explained above. The specification would have to be provided in another text document. The developer of the calling code would have to insert the assertion of the library preconditions whenever he calls the library functions.

Chapter 5

Presentation of Verification Tasks

The CPAclipse plugin for the Eclipse integrated development environment was developed in this thesis. CPAclipse provides a convenient way of access to the features of CPAchecker. CPAclipse allows defining and modifying verification tasks (definded in Chapter 3). The user can invoke the verification of tasks with a single command. Results are displayed in a clear manner and can be accessed depending on which detail of information is needed.

CPAclipse is generally platform independent although individual CPAs might require a specific operating system or system configuration. For example some CPAs are dependent on the MathSAT[1] libraries which can only be supplied for Linux. The development team is currently trying to replace this dependency.

The following sections describe the main components of CPAclipse in the order they occur in the lifecycle of a verification task.

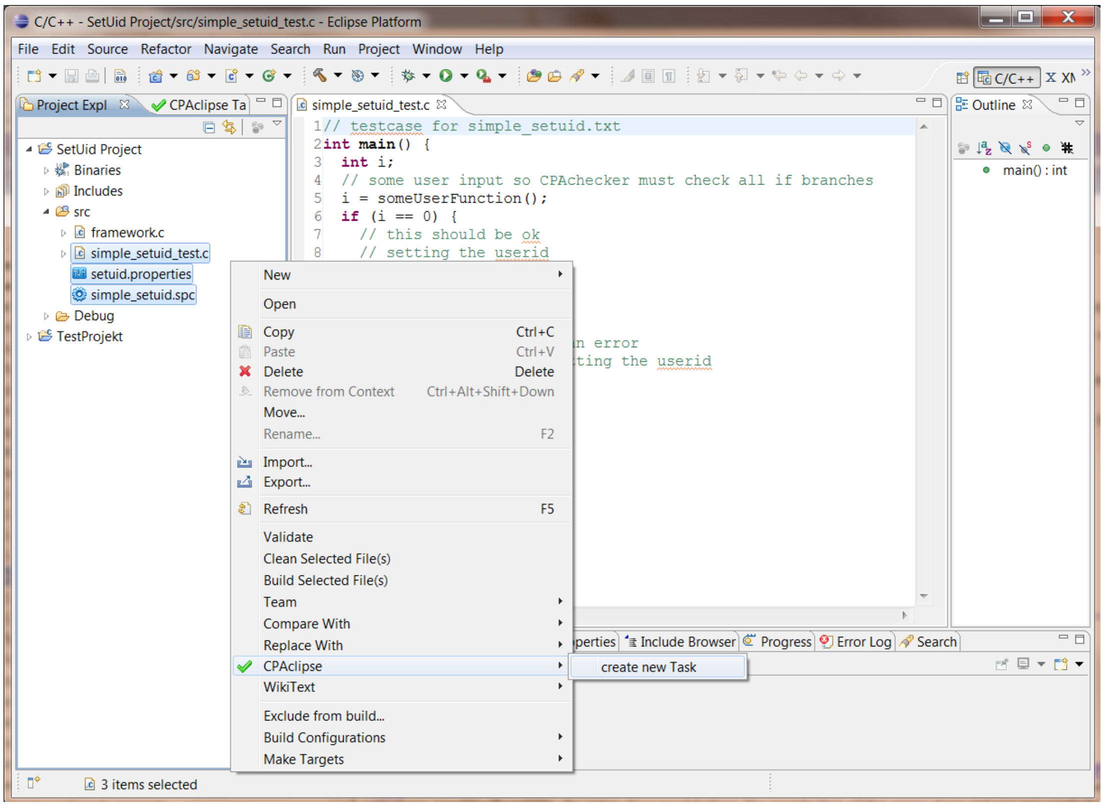
5.1 Generation and Presentation of Verification Tasks

As described in the Chapter Verification Tasks (Chapter 3) the *input information* of a verification task consists of four different parts:

1. The source code that is to be verified
2. The specification defining the behavior that is to be verified
3. An optional set of configuration options
4. A set of results (parts of the output information) from previous analyses that can be reused

Each of the first three parts is represented by one file in CPAchecker. This file might have references to other files (see Section 4.2) but there is one main file. In CPAclipse these files are distinguished by their file extension. Configuration files have the extension ".properties" because they are Java property files. Specification files have the extension ".spc" and are given in the specification language described in Chapter 4 Specification Language. The source code files can have

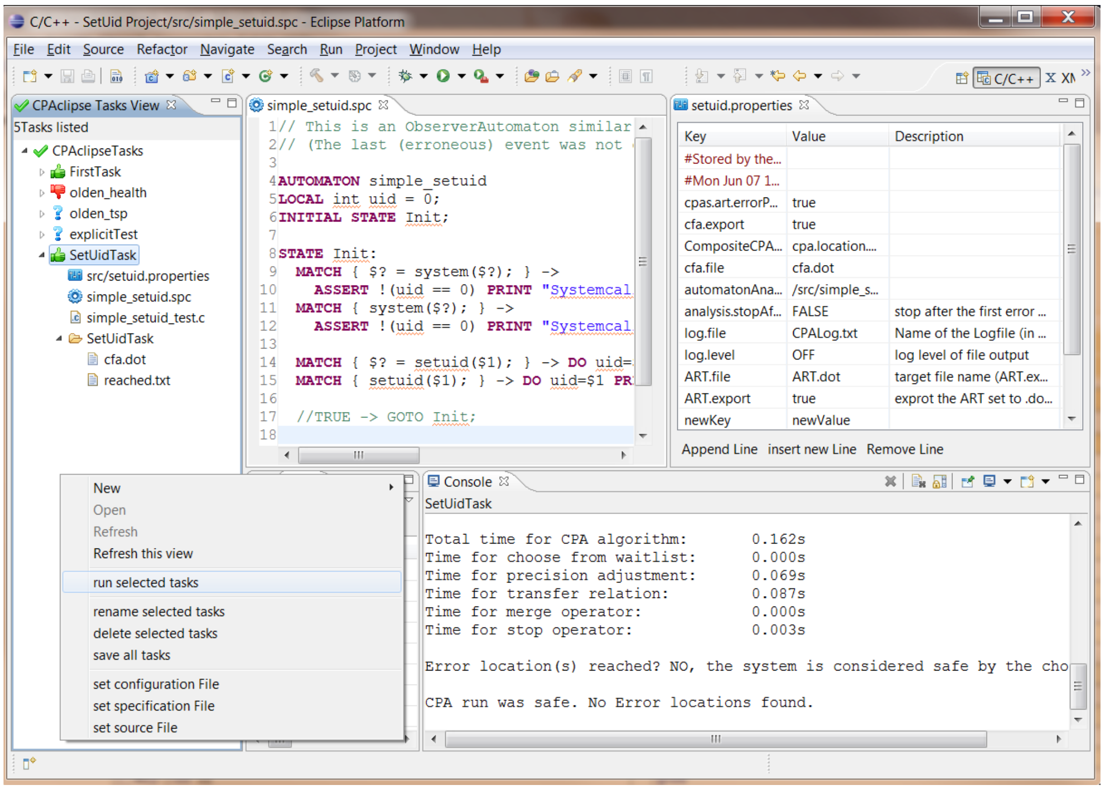
[1] `http://mathsat4.disi.unitn.it`



```
// testcase for simple_setuid.txt
int main() {
  int i;
  // some user input so CPAchecker must check all if branches
  i = someUserFunction();
  if (i == 0) {
    // this should be ok
    // setting the userid
```

```
n error
ting the userid
```

Figure 5.1: Generation of a Verification Task in CPAclipse



```
// This is an ObserverAutomaton similar
// (The last (erroneous) event was not

AUTOMATON simple_setuid
LOCAL int uid = 0;
INITIAL STATE Init;

STATE Init:
  MATCH { $? = system($?); } ->
    ASSERT !(uid == 0) PRINT "Systemcal
  MATCH { system($?); } ->
    ASSERT !(uid == 0) PRINT "Systemcal

  MATCH { $? = setuid($1); } -> DO uid=
  MATCH { setuid($1); } -> DO uid=$1 PR

  //TRUE -> GOTO Init;

```

Key	Value	Description
#Stored by the...		
#Mon Jun 07 1...		
cpas.art.errorP...	true	
cfa.export	true	
CompositeCPA...	cpa.location....	
cfa.file	cfa.dot	
automatonAna...	/src/simple_s...	
analysis.stopAf...	FALSE	stop after the first error ...
log.file	CPALog.txt	Name of the Logfile (in ...
log.level	OFF	log level of file output
ART.file	ART.dot	target file name (ART.ex...
ART.export	true	exprot the ART set to .do...
newKey	newValue	

```
Total time for CPA algorithm:         0.162s
Time for choose from waitlist:        0.000s
Time for precision adjustment:        0.069s
Time for transfer relation:           0.087s
Time for merge operator:              0.000s
Time for stop operator:               0.003s

Error location(s) reached? NO, the system is considered safe by the cho

CPA run was safe. No Error locations found.
```

Figure 5.2: Editors and Verification Invocation in CPAclipse

any extension that is accepted by the Eclipse CDT plugin. The fourth input information component will be discussed in Section 5.3.

CPAclipse provides two ways to generate a verification task. The traditional way is to use the "new CPAchecker Task" wizard. This wizard allows stating the name, the configuration file, the source file and the specification file of the task. If all properties are valid the task can be generated. Because this is a quite cumbersome way of generation, CPAclipse provides a more elegant generation function. If the user selects any one or a combination of the files described above he can select the action "create new task" in the context menu. The action creates a new verification task with the selected files. An example of this function is shown in Figure 5.1.

5.2 Modification and Invocation of Verification Tasks

Verification tasks need to be modified on two levels. The high level modification is used to set components of the task and delete tasks. It is also necessary to save tasks for reuse in later Eclipse sessions. The ability to set for example specification files in tasks is important because of the nature of the generation function explained above. The generation function allows generating a Verification task that is not fully specified. That means the task is still missing one of the three files that is necessary for verification. CPAclipse provides this functions (setting of components and task deletion) via context menus in the "CPAclipse Tasks View". A screenshot of this view is given in Figure 5.2.

The other level of modification concerns the specification and configuration files. For these files CPAclipse provides editors with syntax-highlighting and in case of the "Properties Table Editor" shows a description of the configuration options. Figure 5.2 shows these editors and the "CPAclipse Tasks View".

The Figure also shows how the verification of a set of selected tasks can be invoked. The command to invoke the verification of multiple tasks at once can be found in the context menu of the tree root "CPAclipse Tasks" in the "CPAclipse Tasks View". This command invokes all tasks in strict sequential order. It is planned to introduce parallelism for this command but so far CPAchecker does not support this. As software model checking is potentially time intensive, it is attractive to use the advantages of parallel computing. Because the Verification tasks do not have any dependencies on each other (other than reading from the same files) the introduction of parallelism would be quite easy.

5.3 Presentation of Results

CPAchecker produces four principal kinds of results.

1. The Verification Result as described in Section 3.5
2. The log containing messages that describe the verification process

3. An error path file containing the CFA edges that lead to the error
4. DOT-based graph information files

Each of the files is saved in a Task-specific folder. A link to this folder is displayed with the verification task in the "CPAclipse Tasks View".

The Verification Result has one of the states that were defined in Section 3.5 ($Error\_Found$, No_Error_Found and $Unknown$). As soon as the verification is finished the "CPAclipse Tasks View" shows the Verification Result with an appropriate symbol in front of the Task name. The three symbols can be seen in Figure 5.2.

The log of each verification task is displayed in a separate console view which can also be seen in Figure 5.2. The log contains statistical information and information on the running time of the verification. If the user does not want this information it can be switched off in the CPAclipse preferences dialog.

If the verification finds an error the error path is saved in a text-based file. This file contains textual representations of CFA edges. Each CFA edge has a reference to the source code location that it was generated from. CPAclipse provides an editor to view the error path files and follow the references to the source code locations via hyperlinks. This helps greatly in the task of interpreting and using the information given by the error path.

The fourth kind of output information is DOT-based graph files. These graphs can contain various types of information. For example the generated CFA can be saved in DOT-format. CPAclipse does currently not provide a good means to display this information. It would be desirable to have a function that displays a graphical representation of the DOT-file when the user opens the file. Such a function is in development as Eclipse feature in the dot4zest project[2]. Once this project reaches a stable phase it can be incorporated into CPAclipse to provide better presentation capabilities for graph based results. Then the various possibilities for displaying meta-information with the graphs (as described in Section 3.5) can be implemented easily.

[2] `http://wiki.eclipse.org/Graphviz_DOT_as_a_DSL_for_Zest`

Chapter 6

Conclusion and Further Work

The concept of Verification Tasks and the specification language for software model checking that was developed in this thesis provide intuitive, scalable means to specify the desired behavior of a program. This specification can be used by a model checker to verify the program and to provide helpful information if the program does not fulfill the requirements. The concept of Verification Tasks and the specification language can in principle be used in any model checking tool, or even in other software analysis tools. However especially the specification language has been tailored for the CPAchecker and is deeply integrated into this tool.

One of the main results of this thesis is the insight that errors should not be defined as property of a program location but as property of a program path. Therefore different paths leading to the same location can be distinguished and treated separately in the specification. Previously this was only possible by modifying the source code and introducing a new program location that is only reached by one of the paths. When using the specification language this is not necessary and the concerns of program source code and behavior specification can be separated.

In contrast to specification languages used in other model checkers like SLAM [BR02] our specification language allows to match on any source code element (except comments). The SLAM language concentrates on the calls of API functions. However the SLAM language provides the possibility of executing actions before or after the analysis has processed a function call node in the CFA. In SLAM the code can be executed before a function is called (in the context of the caller) or in the function call (in the context of the callee). In CPAchecker this can only be simulated by using labels in the source code. This feature could be implemented in the specification language in a future project.

Another possibility for future works on the specification language is the inclusion of multiple termination states. At the moment the only automaton state that terminates the analysis is the error state. The specification language could be extended to allow multiple states with this behavior. These could be used to distinguish multiple reasons for termination. This is interesting if the reason for termination is not the finding of an error in the program but some other domain-specific reason. For example the extended language could be used to specify reachability queries.

The Eclipse plugin for CPAchecker that was developed in this thesis allows easy, user-friendly access to the functionality of the model checker. Because CPAchecker is a project under constant development the plugin was designed for flexibility and can be easily readjusted to changes in the model checker.

The possibilities for future work in the plugin are divided in two areas, verification time optimization and user interface improvements. To optimize the execution time of a group of verification tasks they could be executed in parallel threads. As it has been mentioned in Section 3.7 there are no dependencies between different Verification Tasks and therefore their execution can be trivially scheduled in parallel. At the moment this is not possible due to implementation issues in the CPAchecker but it could be optimized in the future.

For improvement of the user interface it would be helpful to have a graphical display module for DOT-files. For example the abstract reachability tree and the control flow automaton are saved as DOT-files by CPAchecker. With a display module for DOT-files the plugin could provide graphic support for analysis of the error path (if one was found) or the abstract reachability tree. These graphs could be annotated with interactive information and hyperlinks. There is a current development project[1] for such a module in the Eclipse community. Once this project reaches a stable result it could be integrated into the plugin.

[1] `http://wiki.eclipse.org/Graphviz_DOT_as_a_DSL_for_Zest`

Bibliography

[BCG+09] Beyer, Dirk ; Cimatti, Alessandro ; Griggio, Alberto ; Keremoglu, M. E. ; Sebastiani, Roberto: Software Model Checking via Large-Block Encoding. In: *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009, Austin (TX), November 15-18)*, IEEE Computer Society Press, Los Alamitos (CA), 2009. – ISBN 978–1–4244–4966–8, S. 25–32

[BCH+04a] Beyer, Dirk ; Chlipala, Adam J. ; Henzinger, Thomas A. ; Jhala, Ranjit ; Majumdar, Rupak: Generating Tests from Counterexamples. In: *Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE 2004, Edinburgh, May 26-28)*, IEEE Computer Society Press, Los Alamitos (CA), 2004. – ISBN 0–7695–2163–0, S. 326–335

[BCH+04b] Beyer, Dirk ; Chlipala, Adam J. ; Henzinger, Thomas A. ; Jhala, Ranjit ; Majumdar, Rupak: The Blast Query Language for Software Verification. In: Giacobazzi, R. (Hrsg.): *Proceedings of the 11th International Static Analysis Symposium (SAS 2004, Verona, August 26-28)*, Springer-Verlag, Berlin, 2004 (LNCS 3148). – ISBN 3–540–22791–1, S. 2–18

[BHJM04] Beyer, Dirk ; Henzinger, Thomas A. ; Jhala, Ranjit ; Majumdar, Rupak: An Eclipse Plug-in for Model Checking. In: *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC 2004, Bari, June 24-26)*, IEEE Computer Society Press, Los Alamitos (CA), 2004. – ISBN 0–7695–2149–5, S. 251–255

[BHT07] Beyer, Dirk ; Henzinger, Thomas A. ; Théoduloz, Grégory: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, W. (Hrsg.) ; Hermanns, H. (Hrsg.): *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, Springer-Verlag, Berlin, 2007 (LNCS 4590). – ISBN 978–3–540–73367–6, S. 504–518

[BHT08] Beyer, Dirk ; Henzinger, Thomas A. ; Théoduloz, Grégory: Program Analysis with Dynamic Precision Adjustment. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008, L'Aquila, September 15-19)*, IEEE

Computer Society Press, Los Alamitos (CA), 2008. – ISBN 978–1–4244–2187–9, S. 29–38

[BK09] Beyer, Dirk ; Keremoglu, M. E.: CPAchecker: A Tool for Configurable Software Verification / School of Computing Science (CMPT), Simon Fraser University (SFU). 2009 (SFU-CS-2009-02). – Forschungsbericht

[BR02] Ball, Thomas ; Rajamani, Sriram K.: The SLAM project: debugging system software via static analysis. In: *Principles of Programming Languages (POPL 2002, Portland, January 16-18)*, 2002, S. 1–3

[CGP99] Clarke, Edmund M. Jr. ; Grumberg, Orna ; Peled, Doron A.: *Model checking.* Cambridge, MA, USA : MIT Press, 1999. – ISBN 0–262–03270–8

[DDH72] Dahl, O. J. (Hrsg.) ; Dijkstra, E. W. (Hrsg.) ; Hoare, C. A. R. (Hrsg.): *Structured programming.* London, UK, UK : Academic Press Ltd., 1972. – ISBN 0–12–200550–3

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *Java^{TM}Language Specification.* 3. Addison-Wesley Professional, 2005 (The Java$^{\text{TM}}$Series)

[GL94] Grumberg, Orna ; Long, David E.: Model checking and modular verification. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), Nr. 3, S. 843–871. – ISSN 0164–0925

[JM09] Jhala, Ranjit ; Majumdar, Rupak: Software model checking. In: *ACM Computing Survey* 41 (2009), Nr. 4

[KF07] Krishnamurthi, Shriram ; Fisler, Kathi: Foundations of incremental aspect model-checking. In: *Transactions on Software Engineering and Methodology* 16 (2007), Nr. 2, S. 7. – ISSN 1049–331X

[KLM$^+$97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina V. ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Bd. 1241, Springer-Verlag, 1997 (Lecture Notes in Computer Science), S. 220–242

[Lyu07] Lyu, Michael R.: Software Reliability Engineering: A Roadmap. In: *FOSE '07: 2007 Future of Software Engineering.* Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2829–5, S. 153–170

[Neu09] Neumann, Peter G.: Risks to the public. In: *ACM SIGSOFT Software Engineering Notes* 34 (2009), Nr. 2, S. 15–24. – ISSN 0163–5948

[NMRW02] Necula, George C. ; Mcpeak, Scott ; Rahul, Shree P. ; Weimer, Westley: CIL: Intermediate language and tools for analysis and transformation of C programs. In: *In International Conference on Compiler Construction*, 2002, S. 213–228

Chapter 7

Appendix

7.1 Grammar of the CPAchecker Specification Language

The grammar of the CPAchecker Specification Language is given as an excerpt from the CUP[1] grammar that is actually used by CPAchecker. To avoid misunderstandings simple symbols like commas and semicolons are written out. The relation between symbols and terminals is explained in the list of terminals. In the grammar all terminals are written in bold type.

```
// Terminal Symbols:
AND, OR, AUTOMATON, OBSERVER, CONTROL, END, LOCAL,
INITIAL, STATE, ERROR, STOP, EXIT, ASSERT, MATCH,
LABEL, CHECK, EVAL, MODIFY, DO, PRINT, GOTO,
TRUE, FALSE, USE_ALL, USE_FIRST,
EXCLAMATION, // !
ARROW, // ->
SEMICOLON, // ;
COMMA, // ,
COLON, // :
OPEN, // (
CLOSE, // )
NEQ, // !=
EQ, // =
EQEQ, // ==
PLUS, // +
MINUS, // -
IDENTIFIER, // any c-style identifier
INTEGER_LITERAL, // an integer (may be negative)
STRING_LITERAL, // a string enclosed in quotes
CURLYEXPR, // a string enclosed in curly brackets
SQUAREEXPR, // a string enclosed in square brackets
INCLUDE; // #include

// Non Terminal Symbols:
initial, automaton, InitDef, Body, LocalDefs,
StateDefs,Transitions, Assertions, Actions,
LocalDef, StateDef, Transition, Assertion,
Action, PrintArguments, ConstantInt, Int,
InnerInt, Bool, Expression

// Operator precedences:
precedence left PLUS, MINUS;
precedence left AND, OR;
precedence left EQEQ, NEQ;
precedence left EXCLAMATION;

initial ::= automaton initial
```

[1] http://www2.cs.tum.edu/projects/cup/

```
              | /* empty */
              ;

automaton ::= OBSERVER AUTOMATON IDENTIFIER Body END AUTOMATON
              | CONTROL AUTOMATON IDENTIFIER Body END AUTOMATON
              | AUTOMATON IDENTIFIER Body END AUTOMATON
               // the CONTROL keyword is optional
              | Assertion SEMICOLON // abbreviated notation
              | INCLUDE IDENTIFIER
              // IDENTIFIER must be a valid filename
            ;

Body      ::= LocalDefs InitDef StateDefs
            ;

InitDef    ::= INITIAL STATE IDENTIFIER SEMICOLON ;

LocalDefs ::= LocalDef LocalDefs
              | /* empty */
              ;

LocalDef  ::= LOCAL IDENTIFIER IDENTIFIER SEMICOLON
            | LOCAL IDENTIFIER IDENTIFIER EQ ConstantInt SEMICOLON
            ;

StateDefs ::= StateDef StateDefs
            | /* empty */
            ;

StateDef  ::= STATE IDENTIFIER COLON Transitions
            | STATE USE_ALL IDENTIFIER COLON Transitions
            | STATE USE_FIRST IDENTIFIER COLON Transitions
            ;

Transitions ::= Transition Transitions
            | /* empty */
            ;

Transition ::= Bool ARROW Assertions Actions GOTO IDENTIFIER SEMICOLON
               | Bool ARROW Assertions Actions ERROR SEMICOLON
               | Bool ARROW Assertions Actions STOP SEMICOLON
              ;

Assertions ::= Assertion Assertions
             | /* empty */
             ;
Assertion  ::= ASSERT Bool

Actions   ::= Action Actions
            | /* empty */
            ;
Action    ::= DO IDENTIFIER EQ InnerInt
            | PRINT PrintArguments
            | MODIFY OPEN IDENTIFIER COMMA STRING_LITERAL CLOSE
            ;

PrintArguments ::= Expression PrintArguments
             | /* empty */
             ;

Int       ::= ConstantInt
            | OPEN Int CLOSE
            | IDENTIFIER
            | InnerInt PLUS InnerInt
            | InnerInt MINUS InnerInt
            ;

InnerInt  ::= Int
            | EVAL OPEN IDENTIFIER COMMA STRING_LITERAL CLOSE
```

```
             ;

ConstantInt ::= INTEGER_LITERAL
             ;

Bool       ::= TRUE
             | FALSE
             | EXCLAMATION Bool
             | OPEN Bool CLOSE
             | InnerInt EQEQ InnerInt
             | InnerInt NEQ InnerInt
             | Bool EQEQ Bool
             | Bool NEQ Bool
             | Bool AND Bool
             | Bool OR Bool
             | MATCH STRING_LITERAL
             | MATCH CURLYEXPR
             | MATCH SQUAREEXPR
             | MATCH LABEL SQUAREEXPR
             | MATCH EXIT
             | CHECK OPEN IDENTIFIER COMMA STRING_LITERAL CLOSE
             | CHECK OPEN STRING_LITERAL CLOSE
             ;

Expression  ::= Int
             | Bool
             | STRING_LITERAL
             | EVAL OPEN IDENTIFIER COMMA STRING_LITERAL CLOSE
             ;
```

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Master-Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 23.8.2010

Alexander von Rhein