Integrated and Automated Abstract Interpretation, Verification and Testing of C/C++ Modules

Jan Peleska*

Department of Mathematics and Computer Science
University of Bremen
Germany
jp@tzi.de

Abstract. Starting from the perspective of safety-critical systems development in avionics, railways and the automotive domain, we advocate an integrated verification approach for C/C++ modules combining abstract interpretation, formal verification and conventional testing. It is illustrated how testing and formal verification can benefit from abstract interpretation results and, vice versa, how test automation techniques may help to reduce the well known problem of *false alarms* frequently encountered in abstract interpretations. As a consequence, verification tools integrating these different methodologies can provide a wider variety of useful results to their users and facilitate the bug localisation processes involved. When applied to C/C++ software, the problems of aliasing, type casts and mixed arithmetic and bit operations have to be handled on the level of constraint generation. We cope with this problem by using a symbolic interpretation method operating on an abstracted memory model. We describe the available tool support developed by the author, his research group and industrial partners.

1 Introduction

1.1 Objectives

In this contribution an integrated approach to static analysis by abstract interpretation, formal verification by model checking and testing is described. The focus of our contribution is on the verification of C/C++ functions and methods (we use the general term *modules* to denote both functions and methods). Module verification[1] has its well-defined place in the development life cycle, and static analysis, testing and – though less frequently used in today's industrial practice

* Partially supported by the BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015B

[1] Following [21], we use the term *verification* for all activities where a development artefact is checked for compliance with respect to a given specification. In particular, reviews, inspections, *formal* verifications, static analyses and testing are verification activities.

– formal verification are recommended techniques for this purpose. From the verification specialists' point of view it is advisable to perform these techniques in an integrated manner:

- Test cases can serve as useful counter examples for violated assertions, thereby supporting the formal verification and static analysis processes,
- static analyses frequently uncover non-functional defects[2] which are unlikely to be detected during functional testing,
- formal verification is the "last resort" when algorithms are too complex to be tested and analysed in an exhaustive way.

To our best knowledge, however, tools supporting these activities in an integrated manner do not exist until today. It is therefore the purpose of this paper to outline how such an integration can be performed and to present the associated tool support developed by author's research group in cooperation with industrial partners. Indeed, it will become apparent that such an integration is also beneficial from the tool builders' point of view:

- As will be outlined in Section 2, both functional and structural testing can be regarded as *reachability problems* as they are typically explored in formal verification by (bounded) model checking.
- Static analysis by abstract interpretation is a powerful means to reduce the state space to be explored for the purpose of test case generation or formal verification.
- Test case generation and the under-approximation techniques used in the constraint solving activities involved are useful for verifying potential errors uncovered by (over approximating) static analyses.

1.2 Background and Motivation: Industrial Safety-Critical Systems Development and the Deployment of Formal Methods

According to the standards [21, 8, 1] the generation of 100% correct software code is not a primary objective in the development of safety-critical systems. This attitude is not unjustified, since code correctness will certainly not automatically imply system safety. Indeed, safety is an *emergent* property [14, p. 138], resulting from a suitable combination of (potentially failing) hardware and software layers. As a consequence, the standards require that

- the contribution of software components to system safety (or, conversely, the hazards that may be caused by faulty software) shall be clearly identified, and
- the software shall be developed and verified with state-of-the art techniques and with an effort proportional to the component's criticality.

[2] In particular, the so-called *runtime errors*, such as division by zero, array bounds violations, out-of-bounds pointers, unintended endless loops etc.

Based on the criticality, the standards define clearly which techniques are considered as appropriate and which effort is sufficient. The effort to be spent on verification is defined most precisely with respect to testing techniques: Tests should (1) exercise each functional requirement at least once, (2) cover the code completely, the applicable coverage criteria (statement, branch, modified condition/decision coverage) again depending on the criticality, (3) show the proper integration of software on target hardware. Task (3) is of particular importance, since analyses and formal verifications on source code level cannot prove that the module will execute correctly on a specific hardware component.

These considerations motivate the main objectives for the tool support we wish to provide:

1. Application of the tool and the results it provides have to be associated clearly with the development phases and artifacts to be produced by each activity specified in the applicable standards.
2. Application of the tool should help to produce the required results – tests, analysis and formal verifications – faster and at least with the same quality as could be achieved in a manual way.

Requirement 1. is obviously fulfilled, since the tool functionality described here has been explicitly designed for the module verification phase, as defined by the standards mentioned above. Requirement 2 motivates our *bug finder* approach with respect to formal verification and static analysis: These techniques should help to find errors more quickly than would be possible with manual inspections and tests alone – finding *all* errors of a certain class is not an issue. As a consequence the tool can be designed in such a way that state explosions, long computation times, false alarms and other aspects of conventional model checkers and static analysis tools, usually leading to user frustration and rejection of an otherwise promising method, simply do not happen: Instead, partial verification results are delivered, and these – in combination with the obligatory tests – are usually much better than what a manual verification could produce within affordable time.

1.3 Related Work

The work presented here summarises and extends results previously published by the author and his research team in cooperation with Verified Systems International GmbH [3, 18, 19, 17].

Many authors point out that the syntactic richness and the semantic ambiguities of C/C++ present considerable stumbling blocks when developing analysis tools for software written in these languages. Our approach is similar to that of [12] in that we consider a simplified syntactic variant – the GIMPLE code – with the same expressive power but far more restrictive syntax than the original language: GIMPLE [11] is a control flow graph representation using 3-address code in assignments and guard conditions. Since the gcc compiler transforms every C/C++ function or method into a GIMPLE representation, this seems to be

an appropriate choice: If tools can handle the full range of GIMPLE code, they can implicitly handle all C/C++ programs accepted by gcc. Therefore we extract type information and GIMPLE code from the gcc compiler; this technique has been described in [15]. In contrast to [12], where a more abstract memory model is used, our approach can handle type casts.

The full consideration of C/C++ aliasing situations with pointers, casts and unions is achieved at the price of lesser performance. In [7, 5], for example, it is pointed out how more restrictive programming styles, in particular, the avoidance of pointer arithmetics, can result in highly effective static analyses with very low rates of false alarms. Conversely it is pointed out in [25] that efficient checks of pointer arithmetics can be realised if only some aspects of correctness (absence of out-of-bounds array access) are investigated. As another alternative, efficient static analysis results for large general C-programs can be achieved if a higher number of false alarms (or alternatively, a suppression of potential failures) is acceptable [9], so that paths leading to potential failures can be identified more often on a syntactic basis without having to fall back on constraint solving methods.

On the level of binary program code verification impressive results have been achieved for certain real-world controller platforms, using explicit representation models [22]. These are, however, not transferable to the framework underlying our work, since the necessity to handle floating point and wide integer types (64 or 128 bit) forbids the explicit enumeration of potential input values and program variable states.

All techniques described in this paper are implemented in the RT-Tester tool developed by the author and his research group at the University of Bremen in cooperation with Verified Systems International GmbH [26]. The approach pursued with the RT-Tester tool differs from the strategies of other authors [7, 5, 25]: We advocate an approach where verification activities focus on small program units (a few functions or methods) and should be guided by the expertise of the development or verification specialists. Therefore the RT-Tester tool provides mechanisms for specifying preconditions about the expected or admissible input data for the unit under inspection as well as for semi-automated stub ("mock-object") generation showing user-defined behaviour whenever invoked by the unit to be analysed. As a consequence, programmed units can be verified immediately – this may be appealing to developers in favour of the *test-driven development* paradigm [4] – and interactive support for bug-localisation and further investigation of potential failures is provided: A debugger supports various abstract interpretation modes (in particular, interval analysis) and the test case generator can be invoked for generating explicit input data for reaching certain code locations indicating the failure of assertions.

With the recent progress made in the field of Satisfiability Modulo Theory [20] powerful constraint solvers are available which can handle different data types, including floating point values and associated non-linear constraints involving transcendent functions. The interplay between path generator, interpreters and solver as handled within the RT-Tester tool has been described in [3]. The solver

implemented in the tool relies on ideas developed in [10] as far as Boolean and floating point constraints are involved, but uses additional techniques and underlying theories for handling linear inequations, bit vectors, strings and algebraic reasoning, see, e. g. [23]. Most methods for solving constraints on interval lattices used in our tool are based on the interval analysis techniques described in [13].

1.4 Overview

In section 2 an overview over the tool architecture and the methods involved is given. The next two sections describe two of the main techniques that are prerequisites for abstract interpretation, property checking and testing: Symbolic interpretation techniques (Section 3) are used to create memory models, symbolically describing the state transitions performed by the UUT along a single path or a whole portion of the code. The constraint generator (Section 4) evaluates the memory model in order to resolve expressions in such a way that the resulting reachability constraints are suitable for the tool's solver component. Section 5 presents a conclusion.

2 Abstract Interpretation, Formal Verification and Testing – an Integrated Approach

2.1 Specification of Analysis, Verification and Test Objectives

In our approach functional requirements of C/C++ modules are specified by means of pre- and post-conditions (Fig. 1). Optionally, additional assertions can be inserted into an "inspection copy" of the module code. The *Unit Under Test (UUT)*[3] is registered by means of its prototype specification preceded by the `@uut` keyword and extended by a `{@pre: ... @post};` block. Pre- and post-conditions are specified as Boolean expressions or C/C++ functions, so that – apart from a few macros like `@pre, @post, @assert` and the utilisation of the method name as place holder for return values – no additional assertion language syntax is required. The pre-condition in Fig. 1, for example, states that the specified module behaviour is only granted if input i is in range $0 \leq i \leq 9$ and inputs x, y satisfy $\exp(y) < x$. The post-condition specifies assertions whose applicability may depend on the input data: The first assertion `globx == globx@pre` states that the global variable `globx` should always remain unchanged by an execution of `f()`. The second assertion (line 9) only applies if the input data satisfies $-10.0 < y \wedge \exp(y) < x$. Alternatively (line 12), the return value of `f()` shall be negative.

It is well-known that pre-/post-condition specifications are considerably facilitated by the optional utilisation of *auxiliary variables* [2, p. 192]: These variables are characterised by the fact that they are never read in control conditions or assignments to non-auxiliary variables. As a consequence, the existence of auxiliary variables and their associated assignments does not change the (untimed)

[3] We use this term in general for any module to be analysed, verified and/or tested.

```
1       double globx;
2       ...
3       @uut double f(double x, double y, int i) {
4         @pre:
5             0 <= i and i <= 9 and exp(y) < x;
6         @post:
7             @assert( globx == globx@pre );
8             if ( -10.0 < y and exp(y) < x ) {
9                @assert( f == 1.0/(x - exp(y)) );
10            }
11            else {
12               @assert( f < 0 );
13            }
14      };
15
```

Fig. 1. Example: Module specification by pre- and post-conditions.

behaviour of the UUT. Assignments can either be directly inserted into the UUT code (so-called *code instrumentation*) or into the UUT specification by way of pre- and post-processing statements: Figure 2 shows an example of the latter variant, where the two previous return values of function `g()` are related to the actual function return: The `@aux` section is used to declare auxiliary variables. The `@preprocess` statements are executed before each call to the UUT, the `@postprocess` statements are executed after the UUT has terminated and the post-condition has been evaluated.

Since module behaviour is not only defined by its input-output relation but also by the sequence of sub-function and method invocations, it is necessary to specify

- the expected number and sequence of sub-function invocations,
- the expected input data to be passed by the UUT to its sub-functions,
- constraints about the sub-function behaviour, depending on the input data it receives.

Sub-functions are specified in the same way as the UUT itself. Using auxiliary variables and associated assignments recording the calls and their parameters, the assertions related to sequencing of sub-function calls can be expressed by means of predicates referring to these auxiliary variables. For test purposes, our system automatically generates *test stubs* (also called *mock objects* in object-oriented settings): These are functions replacing the original sub-functions invoked by the UUT, and showing the specified sub-function behaviour. The utilisation of stubs has the advantage, that exceptional behaviour which rarely occurs in the original sub-function (e. g. report of an arithmetic exception or a hardware error) can easily be simulated in the stub, so that execution of the associated

```
1       @uut double g(double x) {
2         @aux:
3             double z0;
4             double z1 = 0;
5         @preprocess:
6             z0 = z1;
7         @pre:
8             0 < x;
9         @post:
10            @assert( fabs( 1.0 - (g + z1 + z0)/3.0 ) < 0.1 );
11        @postprocess:
12            z1 = g;
13      };
14
```

Fig. 2. Module specification using auxiliary variables.

code sections in the UUT can be triggered in a simple way. For structural testing, the desired coverage can be specified. Currently, we support the coverage criteria required in the standards [21, 8]:

- Statement coverage (C0): Every statement is executed at least once.
- Decision coverage (C1): C0 coverage plus the requirement that every decision is evaluated at least once with result `true` and at least once with result `false`. This is required, for example, for testing avionic software of criticality level B (A = highest criticality level).
- Multiple condition/decision coverage (MC/DC): C1 coverage plus the requirement that every condition in a decision in the module has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown independently to affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. This is required, for example, for testing avionic software of criticality level A.

The specification of pre-/post-conditions and internal assertions, in combination with the optional utilisation of auxiliary variables, allows to specify safety conditions about the module behaviour. As a consequence, the verification goals are represented by reachability problems which are very similar to the structural coverage test goals: If we consider augmented module versions where each safety condition ψ is represented by an auxiliary code branch `if` $\neg\psi$ `then` { `raiseError`(); } located at the appropriate place in the code, a test reaching the `raiseError();`-statement would uncover the violation of ψ and at the same time provide a counter example. Conversely, if this statement can be proven to be "dead code", this proves validity of ψ.

Furthermore, the objective to achieve *functional* test coverage can also be reduced to the problem of achieving *structural* test coverage, that is, it can also be transformed into a set of reachability problems. To illustrate this we consider a typical post-condition pattern

$$Q \equiv \bigwedge_i (C_i(\boldsymbol{v}, \boldsymbol{v}') \Rightarrow Q_i(\boldsymbol{v}, \boldsymbol{v}'))$$

Given variable vector pre-states $\boldsymbol{v}$ and post-states $\boldsymbol{v}'$, this post-conditions states a number of conditions $C_i(\boldsymbol{v}, \boldsymbol{v}')$ about the situations to be distinguished. Depending on the applicable situation $C_i(\boldsymbol{v}, \boldsymbol{v}')$, additional assertions $Q_i(\boldsymbol{v}, \boldsymbol{v}')$ shall also hold. Functional test coverage would now require to create each of the situations $C_i(\boldsymbol{v}, \boldsymbol{v}')$, so that the expected outcome $Q_i(\boldsymbol{v}, \boldsymbol{v}')$ can be checked. Instead of UUT `f`(), we now consider the augmented function $\texttt{f}_{\texttt{aug}}()$ shown in Fig. 3. Obviously, statement coverage of $\texttt{f}_{\texttt{aug}}()$ implies functional coverage of `f`() in the sense exemplified above.

```
1     void f_aug(t1 x1, ..., tn xn) {
2       t r;
3       if ( P(v) ) {
4         // This branch is entered when input data
5         // satisfied pre-condition P(v)
6
7         v0 = v;            // Create copy of pre-states
8         r = f(x1, ...,xn); // Call the UUT
9
10        // Post-state has changed variable vector v,
11        // pre-state is saved in auxiliary variable v0.
12
13        if ( C_1(v0,v) ) {
14          assert( Q_1(v0,v) );
15        }
16        ...
17        if ( C_k(v0,v) ) {
18          assert( Q_k(v0,v) );
19        }
20      }
21    }
22
```

Fig. 3. Branch coverage of `f_aug()` implies functional test coverage of `f()`.

For the static analysis objective *"Absence of run-time errors"* no user-defined specifications are required, since the analysis obligations can be directly extracted from the code. It is possible, however, to choose between *bug finder*

mode and *proof mode*: The former mode only uncovers run-time errors along the module paths which have been investigated in order to reach the specified test coverage and verification goals. Each uncovered run-time error is associated with a test case uncovering the erroneous module state; potential runtime errors for which no test cases could be constructed are not reported. The proof mode tries to prove the absence of *any* runtime error within the module, provided that the specified pre-conditions are met.

2.2 Building Blocks of the Tool Platform

Figure 4 shows the major building blocks of the tool platform which are described in the subsequent paragraphs.

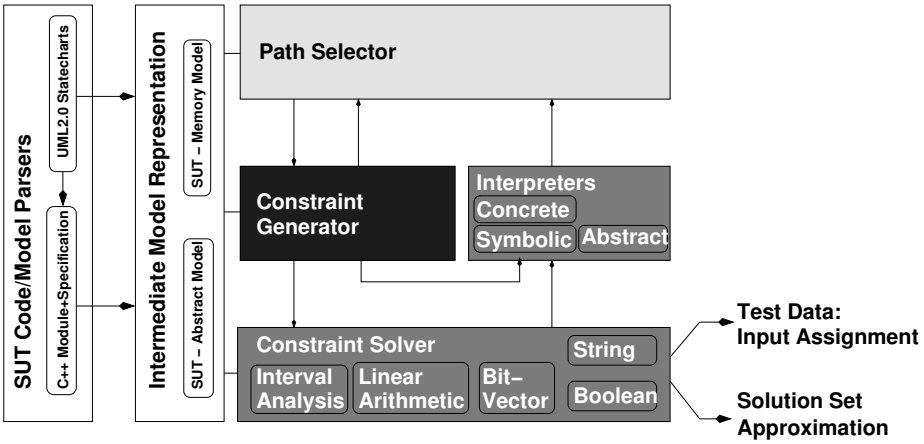

Fig. 4. Building blocks of test automation, static analysis and property verification tool platform.

Intermediate Model Representation and Parser Front-Ends. In order to support various specification formalisms and test paradigms, such as code-based white-box and model-based black-box testing, the system under test and/or its specification are first transformed into an *intermediate model representation (IMR)* which is independent on the concrete SUT code or specification syntax: The IMR consists of a class library allowing to encode hierarchic hybrid transition systems. For testing C/C++ modules, a compilation front-end derived from `gcc` [15] parses the UUT code and generates (1) a control flow graph (CFG) representation in 3-address code of the UUT and (2) a detailed information base supporting queries about types, variables/objects and their sizes. Based on these information, the IMR model of the UUT is instantiated. For model-based test

and verification parsers for UML2.0 and domain-specific languages (railways and automotive) are available.

The CFG representation of C/C++ modules uses GIMPLE syntax [11] which has the same expressiveness as C/C++ but uses a very restricted syntax for expressions. This facilitates the parsing and the IMR generation process, as well as the constraint generation process.

Path Selector. The path selector controls the lazy "on-demand" expansion of the symbolic models described in more detail in Section 3: For a given edge e in the CFG, it constructs paths from the initial CFG node to e and submits them to the constraint generator as sequences or trees of CFG nodes and edges. CFG cycles representing unbounded loops in the UUT code are gradually expanded. Cycles representing simple for-loops with fixed numbers of cycles are identified by means of the interpreters (see below) and directly expanded to their specified limit. With information provided by the abstract interpreters or the solver, the path selector learns about infeasible paths, so that they are never extended. Further details are described in [3].

Interpreters and the Generation of Models. As depicted in Fig. 4, three types of interpreters are used to support the testing, verification and abstract interpretation activities. (1) The *concrete interpreter* evaluates the module with concrete input data, following the rules of the GIMPLE operational semantics, as described in [17]. This interpreter is used to investigate the paths through the GIMPLE module which are covered by given concrete sets of data. If, for example, concrete test data has been generated in order to reach a CFG edge $p \xrightarrow{g}_{CFG} q$, then the concrete interpreter is applied to determine the consecutive transitions $q \longrightarrow_{CFG} \ldots$ to be performed with these data until the module's exit point is reached. (2) The *symbolic interpreter* performs symbolic computations on the CFG, recording the effect of GIMPLE statements by means of predicates in a symbolic memory model, whose details are described in Section 3. This interpreter is the core component for generating the constraints to be solved for the inputs to the module in order to reach a given edge or location in the CFG. Observe that, since the symbolic interpreter does not check whether the constraints recorded along an "interpretation route" through the CFG are satisfiable, the generated computations are a superset of the ones really possible. (3) The *abstract interpreters* evaluate one or more abstractions of the memory model. Starting with (lattice) abstractions of the module's input data, they operate on abstractions of the symbolic memory model. The purpose of this activity is threefold:

- Identification of runtime errors.
- Using over-approximation, an abstract interpreter can find sufficient conditions to prove that a computation "suggested" by path generator and symbolic interpreter is infeasible. Since abstract interpretation can be performed at comparably low cost this is more effective than waiting for the constraint solver to find out that a path cannot be covered.

– Using under-approximation, the abstract interpreters speed up the solution process for non-linear constraints involving floating point variables and transcendent functions.

In order to prove the correctness of interpreters – this is an ongoing activity in the author's research group, but beyond the scope of this paper – it is helpful to analyse the formal rôle of these interpreters: The concrete interpreter obviously generates the concrete transition system representing the behaviour of the GIMPLE module. Hypothetically it could be used to generate a global model for a concrete model checker, but in practice these models would be far too large for any module of realistic complexity. As a consequence, the concrete interpreter unfolds the model in a lazy manner, only along the paths through the CFG where concrete investigations (e. g. for determining the effect of a concrete data set on the module portion to be covered) are required.

The symbolic interpreter unfolds a memory model which implicitly specifies the *power set lattice* over the concrete model. Each state of this power set lattice is represented by a set of pairs (l, s) where l is a location in the module's CFG and s is an interpretation of symbols (variables, pointers, function pointers) in location l. As will become apparent in Section 3, a symbolic memory state *mem* generated during a symbolic computation for a location l defines a constraint ϕ over symbols $x_i \in V$ such that the set of all possible concrete symbol valuations in this state is given by $\{s : V \not\rightarrow D \mid \phi[s(x_1)/x_1, \ldots, s(x_n)/x_n]\}$, that is, the set of valuations where ϕ becomes true.

Each abstract interpreter is instantiated according to the following recipe:

1. For every datatype t in the concrete program component chose a suitable abstraction lattice $(L(t), \sqsubseteq)$, so that a Galois Connection (see, for example [6, p. 201]) $(\mathbf{P}(t), \subseteq) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} (L(t), \sqsubseteq)$ between powerset lattice and abstraction lattice exists.
2. Lift each operation $\Diamond$ defined on t to $L(t)$ by means of the canonic construction $\Diamond_L : L(t) \times L(t) \to L(t)$; $p_1 \Diamond_L p_2 =_{\text{def}} ({p_1}^{\triangleleft} \Diamond_{\mathbf{P}} {p_2}^{\triangleleft})^{\triangleright}$. In this definition, $\Diamond_{\mathbf{P}}$ denotes the canonic lifting of $\Diamond$ to the powerset lattice over t: $a_1 \Diamond_{\mathbf{P}} a_2 =_{\text{def}} \{x_1 \Diamond x_2 \mid x_i \in a_i, i = 1, 2\}$.
3. Having defined all abstraction lattices $L(t)$, lift all Boolean operators $\triangle : t \times t' \to \mathbb{B}$ to $[\triangle] : L(t) \times L(t') \to L(\mathbb{B})$ by

$$p_1[\triangle]p_2 = \begin{cases} \top \text{ if } \{x_1 \triangle x_2 \mid x_1 \in {p_1}^{\triangleleft}, x_2 \in {p_2}^{\triangleleft}\} = \{\texttt{false}, \texttt{true}\} \\ \texttt{false} \text{ if } \{x_1 \triangle x_2 \mid x_1 \in {p_1}^{\triangleleft}, x_2 \in {p_2}^{\triangleleft}\} = \{\texttt{false}\} \\ \texttt{true} \text{ if } \{x_1 \triangle x_2 \mid x_1 \in {p_1}^{\triangleleft}, x_2 \in {p_2}^{\triangleleft}\} = \{\texttt{true}\} \end{cases}$$

4. Based on steps 1 — 3, lift the symbolic state space S_S defined in Section 3 to a lattice representation S_L, together with a Galois Connection $S_S \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} S_L$.
5. Introduce an abstract transition relation $\longrightarrow_L \subseteq S_L \times S_L$ by requiring ($\longrightarrow_{\mathbf{P}}$ denotes the transition relation on the powerset lattice over S_S)

$$\texttt{(a)} \quad \frac{p^{\triangleright\,\triangleleft} \longrightarrow_{\mathbf{P}} p'}{p^{\triangleright} \longrightarrow_L {p'}^{\triangleright}} \qquad\qquad \texttt{(b)} \quad \frac{a^{\triangleleft} \longrightarrow_{\mathbf{P}} p'}{a \longrightarrow_L {p'}^{\triangleright}}$$

This transition relation $\longrightarrow_L$ satisfies the consistency condition

$$(\mathsf{c}) \quad \forall a, a', b \in L : (a \longrightarrow_L a' \wedge b \sqsubseteq a \Rightarrow \exists b' \in L : b \longrightarrow_L b' \wedge b' \sqsubseteq a')$$

Properties (a—c) can be used to identify infeasible transitions on symbolic interpretation level, without having to consult the constraint solver: If abstract interpretation can show that $p^{\triangleright} \not\longrightarrow_L p'^{\triangleright}$, this implies $p^{\triangleright\triangleleft} \not\longrightarrow_{\mathbf{P}} p'$. (For further details, see the lecture notes [16]).

Constraint Generation. The partial symbolic model expanded by the symbolic interpreter according to guidelines of the path selector already contains the constraints to be fulfilled in order to reach a given edge in the CFG on a (set of) pre-defined paths. These constraints, however, still refer to pointer and array expressions which have to be resolved before passing the constraints to the solver. This resolution process is performed by the constraint generator. The resulting constraint ϕ only refers to atomic variables which are derived from the program variables, address offsets and length expressions. The representation of ϕ in conjunctive normal form is structured in such a way that each atomic condition is represented in 3-address code (such as $x < y + z$), as long as it does not involve calls to n-ary functions with $n > 2$ (e. g., $x = f(u, v, w, y)$).

Constraint Solver. The solver handling conditions prepared by the constraint generator has been developed according to the *Satisfiability Modulo Theory (SMT)* paradigm [20]. It uses a combination of techniques for solving partial problems of specific type (e. g., constraints involving bit vector arithmetic, strings, or floating point calculations). For the solution of constraints involving floating point expressions and transcendent functions the solver applies *interval analysis* with sub-paving, bi-partitioning and forward-backward constraint propagation as main techniques [13]. Given a constraint Φ and its solution set $\mathbb{S}$, the solver constructs a sub-paving P, that is, a collection of interval vectors whose union C is a subset of $\mathbb{S}$, that is, and under-approximation. As a consequence, any vector of variable values taken from P is a solution of Φ. Alternatively, the solver can construct sub-pavings as over-approximations P', such that the union C' of P'-interval vectors satisfies $\mathbb{S} \subseteq C'$. This is used for the generation of boundary value and robustness tests, where Φ represents a pre-condition, and we are interested in finding values close to the boundary (inside or outside) of $\mathbb{S}$. To speed up the process of finding P, forward-backward constraint propagation is used to contract interval vectors possessing non-empty intersection with both $\mathbb{S}$ and its complement. Additionally, we use learning strategies as described in [10, 3], where also more details on the solver can be found.

3 Memory Model and Symbolic Interpretation

3.1 Memory Model

As a consequence of the aliasing problems of C/C++ it may be quite complex to determine the valuation of a variable in a given module state: the memory

location associated with the variable may have been changed not only by direct assignments referring to the variable name, but also indirectly by assignments to de-referenced pointers and memory copies to areas containing the variable. Therefore we introduce a memory model that allows us to identify the presence of such aliasing effects with acceptable effort. Computations are defined as sequences of memory configurations, and the memory areas affected by assignments or function/method executions are specified by means of base addresses, offsets and physical length of the affected area. Moreover, the values written to these memory areas are only specified *symbolically* by recording the value-defining expression (e. g. right-hand side of an assignment or output parameter of a procedure call) without resolving them to concrete or abstract valuations. This motivates the term *symbolic interpretation*. Global, static and stack variables `x` induce base addresses `&x` in the data and stack segment, respectively. Dynamic memory allocation (`malloc(), new ...`) creates new base addresses on the heap. A memory configuration *mem* consists of a collection of *memory items*, each item m specified by base address, offset, length and and value expression (Fig. 5). Since some statements will only conditionally affect a memory area, it is necessary to associate memory items with constraints specifying the conditions for the item's existence.

$$\boxed{m.v_0 \mid m.v_1 \mid m.a \mid m.t \mid m.o \mid m.l \mid m.val \mid m.c}$$

$m.v_0$ First computation step number where m is valid
$m.v_1$ Last computation step number where m is valid or ∞ for items valid beyond the actual computation step
$m.a$ Symbolic base address
$m.t$ Type of specified value $m.val$
$m.o$ Start offset from base address in bits, where value is stored
$m.l$ Offset from base address to first bit following the stored value, so $m.l - m.o$ specifies the bit-length of the memory location represented by the item
$m.val$ Value specification
$m.c$ Validity constraint

Fig. 5. Structure of a memory item m.

Symbolic computations – that is, sequences of memory configurations related by transition relations – are recorded as *histories*, in order to reduce the required storage space: Memory items are associated with a validity interval $[m.v_0, m.v_1]$ whose boundaries specify the first and last computation step where the item was a member of the configuration.

Example 1. Suppose that variables `float x, y, z;` are defined in the stack frame of the UUT on a 32-bit architecture, and the current computation step n performs

an assignment $\mathtt{x} = \mathtt{y} + \mathtt{z}$. This leads to the creation of a new memory item

$$m =_{\text{def}} \boxed{n \mid \infty \mid \&x \mid \mathtt{float} \mid 0 \mid 32 \mid y_n + z_n \mid \mathtt{true}}$$

Item m is first valid from step n on, and has not yet been invalidated by other writes affecting the memory area from start address $\&x$ to $\&x{+}31$, so $m.v_1 = \infty$. (Example 3 below shows the effect of C/C++ statements on the invalidation of memory items.) The value depends on the valuation of y and z, taken in step n. This is denoted by the version index n in the value expression $y_n + z_n$. □

For the representation of large memory areas carrying identical or interdependent values it is useful to admit additional bound parameters in the offset, value and constraint specifications:

$$m_{p_0,\ldots,p_k} = \boxed{v_0 \mid v_1 \mid a \mid t \mid o(p_0,\ldots,p_k) \mid l(p_0,\ldots,p_k) \mid val(p_0,\ldots,p_k) \mid c(p_0,\ldots,p_k)}$$

defines a *family of memory items* by means of the definition

$$\begin{aligned} m_{p_0,\ldots,p_k} =_{\text{def}} \{m' \mid\ & m'.v_0 = v_0 \wedge m'.v_1 = v_1 \wedge m'.a = a \wedge m'.t = t \wedge \\ & (\exists p'_0,\ldots,p'_k : m'.o = o[p'_0/p_0,\ldots,p'_k/p_k] \wedge \\ & m'.l = l[p'_0/p_0,\ldots,p'_k/p_k] \wedge \\ & m'.val = val[p'_0/p_0,\ldots,p'_k/p_k] \wedge \\ & m'.c = c[p'_0/p_0,\ldots,p'_k/p_k])\} \end{aligned}$$

Example 2. Suppose that array `float a[10];` is defined in the stack frame of the UUT on a 32-bit architecture, and is currently represented by a family of memory items

$$m_p =_{\text{def}} \boxed{n \mid \infty \mid \&a[0] \mid \mathtt{float} \mid 32 \cdot p \mid 32 \cdot p + 32 \mid \mathtt{sinf}((float)p) \mid 0 \le p \wedge p < 10}$$

Family m specifies one memory item for each $p \in \{0,\ldots,9\}$, each item located at a p-dependent offset from the base address $\&a[0]$ and carrying a p-dependent value. □

More formally, the symbolic interpretation state space S_S of a module P is defined as

$S_S =_{\text{def}} N(P) \times \mathbb{N}_0 \times M$

$N(P) =_{\text{def}}$ Nodes of P's GIMPLE control flow graph

$M =_{\text{def}} \mathit{dataSegment} \times \mathit{heapSegment} \times \mathit{stackSegment}$

$\mathit{dataSegment} =_{\text{def}} \mathit{M\text{-}Item}^*$

$\mathit{heapSegment} =_{\text{def}} \mathit{M\text{-}Item}^*$

$\mathit{stackSegment} =_{\text{def}} \mathit{stackFrame}^*$

$\mathit{stackFrame} =_{\text{def}} \mathit{M\text{-}Item}^*$

$\mathit{M\text{-}Item} =_{\text{def}} \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\}) \times \mathit{BaseAddress} \times$
$\mathit{Types} \times \mathit{Offset} \times \mathit{OffsetPlusLength} \times \mathit{Value} \times \mathit{Constraint}$

$\mathit{BaseAddress} =_{\text{def}} \mathit{String}$

$\mathit{Offset} =_{\text{def}} \mathit{OffsetPlusLength} =_{\text{def}} \mathit{Value} =_{\text{def}} \mathit{Constraint} =_{\text{def}} \mathit{Expr}(\mathit{Sym} \times \mathbb{N}_0)$

$\mathit{Sym} =_{\text{def}}$ Symbols of P plus parameters for families of memory items

Each symbolic state consists of a triple $(\mathit{node}, n, \mathit{mem})$ where node is a node in the GIMPLE control flow graph representing the current "program counter state" of the symbolic execution, n serves as a computations step counter and mem is the current memory configuration. The collection of memory items generated so far is structured according to their allocation in the data segment, heap or stack, respectively. The stack is further sub-divided into frames, so that the validity of stack variables during their associated function executions can be clearly specified.

For the symbolic specification of offsets, values and constraints GIMPLE expressions over symbols from the module P, that is, program variables or object attributes are used. In addition to that, these expressions may refer to parameters defining families of memory items, as illustrated in Example 2. Each variable symbol is associated with a version identifier: If x has current version n and the next computation step processes an assignment $x = x + 1$; this generates a new memory item for version x_{n+1} with value expression $x_n + 1$.

3.2 Symbolic Interpretation

Symbolic interpretation (denoted below by transition relation $\longrightarrow_G$, "G" standing for "GIMPLE operational semantics") is performed according to rules of the pattern

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2}{(n_1, n, mem) \longrightarrow_G (n_2, n+1, mem')},$$

so a transition can be performed on symbolic level whenever a corresponding edge exists in the control flow graph ($\xrightarrow{g}_{CFG}$ denotes the edge-relation in the module's CFG, with guard condition g as label). It may turn out, however, on

$\sigma : Symbols \times M \to M\text{-}Item^*$	Depending on the current memory state mem, $\sigma(x, mem)$ maps a symbol x to the stack frame, global data or heap section where x is associated with.
$\tau : Symbols \times M \to Symbols$	Depending on the current memory state m, $\tau(x, m)$ maps a variable symbol x to its type.
$\beta : Selectors \to BaseAddress$	Maps a selector to its base address.
$\omega : Selectors \to Expr$	Maps a selector to its offset expression.
$\texttt{bitsizeof} : Symbols \to \mathbb{N}$	Takes a type symbol and returns its length in bits.
$\texttt{bit} : Expr \times \mathbb{N}_0 \not\to \{0, 1\}$	$\texttt{bit(e, b)}$ returns the value of the bth bit of expression e.

Fig. 6. Auxiliary functions used in the symbolic interpretation algorithms in Section 3.2.

abstract or concrete interpretation level, that such a transition is *infeasible* in the sense that no valuation of inputs exists where the constraints of all memory items involved evaluate to `true`. Informally speaking, a statement changing the memory configuration is processed according to the following steps: (1) For every base address and offset possibly affected by the statement, create a new memory item m', to be added to the resulting configuration. (2) For each new item m' check which existing items m may be invalidated: Invalidation occurs, if m' refers to the same base address as m and the data area of m' covers (i. e., has a non-empty intersection with) that of m. (3) For each invalidated item m create new ones m'' specifying what may still remain visible of m: m'' equals to m if m' does not exist at all (i. e., constraint $m'.c$ evaluates to `false`), or m' and m do not overlap. Moreover, m'' specifies the resulting value representation of m in memory for the situation where m' and m only partially overlap.

To explain the effect of symbolic transitions on the state space S_S more formally, we present three transition rules explaining stack variable definition, assignment to a variable and assignment to a de-referenced pointer. In the definitions and algorithms involved some auxiliary functions are involved which are defined in Fig. 6.

(1) A *stack variable definition*, $n_1 =_{\text{def}}$ `typex x;`, performed in computation step n, only affects the current stack frame. Value expression `Undef` marks that the value is still undefined. The new memory item is only valid if the guard condition g, evaluated according to the memory configuration of computation step $n - 1$ is `true`.

$m := (n, \infty, \&x, \texttt{typex}, 0, \texttt{bitsizeof(typex)}, \texttt{Undef}, (g, n-1))$;
$mem' := (mem.data, mem.heap, front(mem.stack) \frown \langle last(mem.stack) \frown \langle m \rangle \rangle$;

(2) The effect of an *assignment to a stack or global variable*, $n_1 =_{\text{def}}$ `sel = expr;` affects the current stack frame or the global data segment. In this assignment

procedure $up_{=}(sel : Selectors;\ expr : Expr;\ n : \mathbb{N}_0;\ g : Expr;\ \textbf{inout}\ mem : M)$
  $m'.v_0 := n;$
  $m'.v_1 := \infty;$
  $m'.a := \&\beta(sel)$
  $m'.t := \tau(sel, mem)$
  $m'.o := (\omega(sel), n-1)$
  $m'.l := (\omega(sel) + \texttt{bitsizeof(sel)}, n-1)$
  $m'.val := (expr, n-1)$
  $m'.c := (g, n-1)$
  $up(m', n, mem);$
end

Fig. 7. Effect of normal assignments on history of memory items.

expression `sel` denotes an arbitrary *selector*, that is an identifier of an atomic variable, structure component, array element or mixed structure/array identifier, such as `a.b[i][j].c.d[k]`. The new memory state mem' is specified by procedure call

$$up_{=}(sel, expr, n+1, g, mem);\ mem' := mem;$$

and its in-out-parameter mem. Procedure $up_{=}()$ (Fig. 7) specifies (a) how a new memory item m' is created, carrying the right-hand side expression as its value and the CFG guard condition as validity constraint and (b) which memory items m have to be invalidated due to the new assignment, possibly leading to the creation of "replacements" for these m involving new constraints. The details of this invalidation/creation process are specified in procedure $up()$ (Fig. 8).

In the loop processed in procedure $up()$, new memory item m'' captures the situation where either m' is infeasible (i. e. $\neg m'.c$) or the address range of m' does not affect m. The definition of the memory item family m''_b in this specification handles the "worst case" of an assignment, where only one or more, but not all bits of an existing memory item m are overwritten. This happens, for example, when working with C/C++ unions or bit-vector operations where selected bits of an integer variable can be manipulated. Item family m'''_b specifies which of the original bits from m are left unchanged by such an operation. Fortunately, it can be decided in many situations that m'''_b does not exist, so that the invocation of bit-vector decision procedures can be avoided. Assume, for example, that all array indexes involved in expressions `a.b[i]` and `a'.b'[i']` are within range. Then an assignment to `a.b[i]` is in conflict with `a'.b'[i']` if and only if $\texttt{a} = \texttt{a}' \wedge \texttt{b} = \texttt{b}' \wedge \texttt{i} = \texttt{i}'$. In this case, `a'.b'[i']` is completely overwritten. Further observe that the offset expressions $\omega(sel)$ used in procedure $up_{=}()$ are constants if the selector does not involve array indexes or only indexes which are constant.

procedure $up(m' : \textit{M-Item};\ n : \mathbb{N}_0;\ \textbf{inout}\ mem : M)$
begin
  $h := \sigma(m'.a, mem)$;
  $u := \langle\ \rangle$;
  for $m = last(h)$ **downto** $head(h)$ **do**
    if $(m.v_1 = \infty \wedge m'.a = m.a)$ **then**
      $m.v_1 := n - 1$;
      $c'' := m.c \wedge (\neg m'.c \vee m'.l \leq m.o \vee m.l \leq m'.o)$
      $m'' := (n, \infty, m.a, m.t, m.o, m.l, m.val, c'')$;
      $c''' := m.c \wedge m'.c \wedge 0 \leq b \wedge b < \texttt{bitsizeof}(\texttt{m.t}) \wedge$
            $(b < m'.o - m.o \vee m'.l - m.o \leq b)$;
      $m'''_b := (n, \infty, m.a, \texttt{Bit}, m.o + b, m.o + b + 1, \texttt{bit}(\texttt{m.val}, \texttt{b}), c''')$;
      $u := \langle m'', m'''_b \rangle \frown u$;
    endif
  enddo
  $h := h \frown u \frown \langle m' \rangle$;
end

Fig. 8. Effect of new memory item m' on memory items $m \in M$.

Example 3. A stack declaration `int a[10];` followed by assignments `a[i] = m + n; a[j] = 0;` is represented in GIMPLE as

```
1     int a[10];
2     i_0 = i;
3     D_4151 = m + n;
4     a[i_0] = D_4151;
5     j_1 = j;
6     a[j_1] = 0;
```

After having processed lines 1 — 6, the associated computation results in the following history of memory items:

$$m^1_p = (1, 3, \&\texttt{a[0]}, 32 \cdot p, 32 \cdot p + 32, \texttt{int}, \texttt{Undef}, 0 \leq p \wedge p < 10)$$

$$m^2 = (2, \infty, \&\texttt{i\_0}, 0, 32, \texttt{int}, \texttt{i}_1, \texttt{true})$$

$$m^3 = (3, \infty, \&\texttt{D\_4151}, 0, 32, \texttt{int}, \texttt{m}_2 + \texttt{n}_2\texttt{true})$$

$$m^4_p = (4, 5, \&\texttt{a[0]}, 32 \cdot p, 32 \cdot p + 32, \texttt{int}, \texttt{Undef}, 0 \leq p \wedge p < 10 \wedge p \neq i\_0_2)$$

$$m^5 = (4, 5, \&\texttt{a[0]}, 32 \cdot i\_0_2, 32 \cdot i\_0_2 + 32, \texttt{int}, D\_4151_3, 0 \leq i\_0_2 \wedge i\_0_2 < 10)$$

$$m^6 = (5, \infty, \&\texttt{j\_1}, 0, 32, \texttt{int}, \texttt{j}_4, \texttt{true})$$

$$m^7_p = (6, \infty, \&\texttt{a[0]}, 32 \cdot p, 32 \cdot p + 32, \texttt{int}, \texttt{Undef}, 0 \leq p \wedge p < 10 \wedge p \neq i\_0_2 \wedge p \neq j\_1_5)$$

$$m^8 = (6, \infty, \&\texttt{a[0]}, 32 \cdot i\_0_2, 32 \cdot i\_0_2 + 32, \texttt{int}, D\_4151_3,$$
$$0 \leq i\_0_2 \wedge i\_0_2 < 10 \wedge i\_0_2 \neq j\_1_5)$$

$$m^9 = (6, \infty, \&\texttt{a[0]}, 32 \cdot j\_1_5, 32 \cdot j\_1_5 + 32, \texttt{int}, 0, 0 \leq j\_1_5 \wedge j\_1_5 < 10)$$

This example illustrates how memory items are invalidated by consecutive writes to related memory areas: For example, the execution of statements 1 — 4 results in $m_p^1.v_1 = 3$, since the write in statement 4 also affects the memory area starting at &a[0]. Items m_p^4 specifies "what is left of" m_p^1 after execution of statement 4. □

---

procedure $up_{=_p}(p : Symbols;\ expr : Expr;\ n : \mathbb{N}_0;\ g : Expr;$ **inout** $mem : M)$
begin
  $h := \sigma(p, mem)$;
  for $m' = last(h)$ **downto** $head(h)$ **do**
    if $m'.a = \&p \wedge m'.v_1 = \infty$ **then**
      $pl := \xi(m'.val, mem)$
      foreach $m'' \in pl$ **do**
        $m.a :=$ Base address part of address expression m".val;
        $m.v_0 := n$;
        $m.v_1 := \infty$;
        $m.o :=$ Offset part of address expression m".val;
        $m.l := m.o +$ `bitsizeof`$(\tau(*p, mem))$;
        $m.val = (expr, n)$;
        $m.c := (g, n) \wedge m'.c \wedge m''.c$;
        $up(m, n, mem)$;
      enddo
    endif
  enddo
end

Fig. 9. Effect of assignments to de-referenced pointers on history of memory items.

(3) An *assignment to a de-referenced pointer*, $n_1 =_{\text{def}}$ `*p = expr;` may affect the data segment, heap or stack, depending on the potential target addresses `p` points to. The details are specified by function $up_{=_p}$ (Fig. 9).

$up_{=_p}(p, expr, n, mem, g)$
$mem' := mem$;

$up_{=_p}$ loops over all possible pointer valuations represented by memory items m'. For each valid item, a list pl of all possible pointer targets is generated, using auxiliary function $\xi()$ (Fig.10): Depending on the possible valuations of the address expression specified in $m.val$, p may point to one or more locations in stack, data segment or heap. For each of these possible situations, the items m'' in pl contain value expressions consisting of base addresses and offsets. For example, if $\mathtt{p} = \mathtt{q} + \mathtt{i} \wedge \mathtt{q} = \&\mathtt{z} + \mathtt{k}$, then ξ returns an item with value expression $\&z + k + i$ in its list. The effect of each new item on the invalidation of existing

```
function ξ((expr, n) : Expr × ℕ0; mem : M) : M-Item*
  m := (., ., ., ., ., ., (expr, n), true);
  el := ⟨m⟩;
  while have m-item m′ in el with unresolved m′.val do
    m′ := next m-item in el with unresolved m′.val;
    x := next unresolved identifier from m′.val;
    h := σ(x, mem);
    for m″ := last(h) downto head(h) do
      if m″.a = β(x) ∧ m″.v0 ≤ n ≤ m″.v1 then
        val1 := m′.val;
        In val1: exchange each occurrence of x by m″.val;
        el := el ⌢ ⟨(., ., ., ., ., ., val1, m′.c ∧ m″.c)⟩;
      endif
    enddo
    erase m′ from el;
  enddo
  ξ := el;
end
```

Fig. 10. Function ξ finds list of base addresses and offsets potentially associated with a pointer.

items and creation of new ones is performed again as specified by *up*() and explained above.

3.3 Optimisation Through Abstract Interpretation

The symbolic interpretation process as described above does not investigate the feasibility of the memory items associated with a configuration. This can either be done after constraint generation (see Section 4) by the solver or by more efficient, though incomplete, abstract interpretations. To this end, we use interval analysis in order to calculate ranges of possible variable values and pointer target addresses. These interval interpretations are used to check whether constraints $m.c$ of memory items m evaluate to `false` in the interval lattice valuation of Boolean expressions. Since the interval interpretation is an over-approximation, a `false`-valuation of $m.c$ in the interval lattice implies that no concrete valuation of $m.c$ could evaluate to `true` either. As a consequence, m can be immediately dropped, without having to consult the solver.

4 Constraint Generation

As we have seen in the previous section, the guard conditions to be fulfilled in order to cover a specific path or a sub-graph of a module's CFG are already

encoded in the memory items associated with the symbolic memory configurations involved. The most important task for the constraint generator is now to resolve the value components of the memory items involved, so that the resulting expressions are free of pointer and array expressions, and are represented in an appropriate format for the solver.

Example 4. Let us extend Example 3 by two additional statements

```
7 D_4160 = a[i_0];
8 if ( D_4160 < 0 ) { ...(*)... }
```

and suppose we wish to reach the branch marked by (*). The constraint generator now proceeds as follows: (1) Initialise constraint Φ as $\Phi := \texttt{D\_4160} < 0$.

(2) Resolve D_4160 to a[i_0], as induced by the memory item resulting from the assignment in line 7. Since a[i_0] is an array expression, we have to resolve it further, before adding the resolution results to Φ.

(3) $\texttt{a}_7[\texttt{i\_0}_7]$ matches with items m_p^7, m^8, m^9 for a and m^2 for i_0 in Example 3, since the other items with base address &a[0] are already outdated at computation step 7; this leads to resolutions

$$\begin{aligned}\Phi := \Phi \wedge ((&\texttt{D\_4160} = \texttt{Undef} \wedge \texttt{i\_0}_7 = \texttt{p} \wedge 0 \leq \texttt{p} \wedge \texttt{p} < 10 \wedge \texttt{p} \neq \texttt{i\_0}_2 \wedge \texttt{p} \neq \texttt{j\_1}_5) \vee \\ &(\texttt{D\_4160} = \texttt{D\_4151}_3 \wedge \texttt{i\_0}_7 = \texttt{i\_0}_2 \wedge 0 \leq \texttt{i\_0}_2 \wedge \texttt{i\_0}_2 < 10 \wedge \texttt{i\_0}_2 \neq \texttt{j\_1}_5) \vee \\ &(\texttt{D\_4160} = 0 \wedge \texttt{i\_0}_7 = \texttt{j\_1}_5 \wedge 0 \leq \texttt{j\_1}_5 \wedge \texttt{j\_1}_5 < 10)) \wedge \\ &\texttt{i\_0}_7 = \texttt{i\_0}_2 \wedge \texttt{i\_0}_2 = \texttt{i}_1\end{aligned}$$

Observe that at this stage Φ has been completely resolved to atomic data types: The references to array variable a have been transformed into offset restrictions (expressions over $\texttt{i\_0}_7, \texttt{i\_0}_2, \texttt{j\_1}_5, \ldots$), and the array elements involved (in this example a[i_0]) have been replaced by atomic variables representing their values (D_4160). References to C-structures would be eliminated in an analogous way, by introducing address offsets for each structure component and using atomic variables denoting the component values.

Further observe that we have already eliminated the factors 32 in Φ, initially occurring in expressions like $32 \cdot \texttt{i\_0}_7 = 32 \cdot \texttt{j\_1}_5$. These factors are only relevant for bit-related operations; for example, if an integer variable is embedded into a C-union containing a bit-field as another variant, and a memory item corresponding to the integer value is invalidated by a bit operation.

(4) Prepare the constraint for the solver: Following the restrictions for admissible constraints described in [10], our solver requires some pre-processing of Φ: (a) Inequalities like $\texttt{i\_0}_2 \neq \texttt{j\_1}_5$ are replaced by disjunctions involving $<, >$, e. g. $\texttt{i\_0}_2 < \texttt{j\_1}_5 \vee \texttt{i\_0}_2 > \texttt{j\_1}_5$. (b) Inequalities $a < b$ are only admissible if a or b is a constant. Therefore atoms like $\texttt{i\_0}_2 < \texttt{j\_1}_5$ are transformed with the aid of *slack variables* s, so that non-constant symbols are always related by equality. For example, the above atom is transformed into $\texttt{i\_0}_2 + \texttt{s} = \texttt{j\_1}_5 \wedge 0 < \texttt{s}$. (c) Three-address-code is enforced, so that – with the exception of function calls $\texttt{y} = \texttt{f}(\texttt{x}_0, \ldots, \texttt{x}_\texttt{n})$ and array expressions $\texttt{y} = \texttt{a}[\texttt{x}_1] \ldots [\texttt{x}_\texttt{n}]$ – each atom refers to at most 3 variables. Since the introduction of slack variables may lead to four variables in an expression originally expressed with three symbols only, auxiliary variables are needed to reinstate the desired three-address representation.

For example, $x + y < z$ leads to $x + y = z + s \wedge s < 0$ which is subsequently transformed into $aux = z + s \wedge x + y = aux \wedge s < 0$. (d) The constraint is transformed into conjunctive normal form CNF. Constraint Φ in this example already indicates a typical problem to be frequently expected when applying the standard CNF algorithm: Some portions of Φ resemble a disjunctive normal form. This is caused by the necessity to consider alternatives – that is, $\vee$-combinations – of memory items, where the validity of each item is typically specified by a conjunction. As a consequence, the standard CNF algorithm may result in a considerably larger formula. Therefore we have implemented both the standard CNF algorithm and the Tseitin algorithm [24] as an alternative, together with a simple decision procedure indicating which algorithm will lead to better results.

5 Conclusion

We have described an integrated approach for automated testing, static analysis by abstract interpretation and formal verification by model checking (reachability analysis). The techniques described have been explicitly designed for the verification of C/C++ modules. To cope with the aliasing problems of C/C++, a memory model for symbolic interpretation of address values, offsets, lengths and values of memory valuations has been described. The combinatorial complexity of symbolic memory interpretation is considerably reduced by means of lock-step abstract and symbolic interpretation, using the abstract interpretation for *a priori* elimination of infeasible symbolic states. The tasks of functional and structural testing have been reduced to problems of reachability analysis. To cope with constraints involving all C/C++ data types, including bit vector operations, type casts, large integer ranges and floating point variables, an SMT (Satisfiability Modulo Theory) solver is used which handles floating point variables and transcendent functions by means of interval analysis.

Acknowledgements. I would like to express my gratitude to Willem-Paul de Roever, for his considerable support during various stages of an interleaved academic and industrial career. Willem-Paul's advice – both scientific and personal – has always been highly constructive and stimulating.

References

1. *IEC 61508 Functional safety of electric/electronic/programmable electronic safety-related systems.* International Electrotechnical Commission, 2006.
2. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, Berlin Heidelberg New York, 1991.
3. Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test automation for hybrid systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.
4. Kent Beck. *Test-Driven Development.* Addison-Wesley, 2003.

5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. MinÃ©, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS, December 6–8 2006. Springer, Berlin. (to appear).
6. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
7. Bruno Blanchet et. al. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. AE. Mogensen et al., editor, *The Essence of Computation*, volume 2566, pages 85–108, 2002.
8. European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels, 2001.
9. Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna - a static model checker. In *Proceedings of 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Bonn, Germany, 2006.
10. Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 2007.
11. GCC, the GNU Compiler Collection. The GIMPLE family of intermediate representations. See `http://gcc.gnu.org/wiki/GIMPLE`.
12. Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379, Paris, France, January 2005. Springer.
13. Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis*. Springer-Verlag, London, 2001.
14. Nancy G. Leveson. *Safeware*. Addison-Wesley, 1995.
15. Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen, May 2007.
16. Jan Peleska and Helge Löding. *Static Analysis By Abstract Interpretation*. University of Bremen, Centre of Information Technology, 2008. available under `http://www.informatik.uni-bremen.de/agbs/lehre/ws0708/ai/saai_script.pdf`.
17. Jan Peleska and Helge Löding. Symbolic and abstract interpretation for c/c++ programs. In *Proceedings of the 3rd intl Workshop on Systems Software Verification (SSV08)*, Electronic Notes in Theoretical Computer Science. Elsevier, February 2008.
18. Jan Peleska, Helge Löding, and Tatiana Kotas. Test automation meets static analysis. In Rainer Koschke, Karl-Heinz Rödiger Otthein Herzog, and Marc Ronthaler, editors, *Proceedings of the INFORMATIK 2007, Band 2, 24. - 27. September, Bremen (Germany)*, pages 280–286.
19. Jan Peleska and Cornelia Zahlten. Integrated automated test case generation and static analysis. In *Proceedings of the QA+Test 2007 International Conference on QA+Testing Embedded Systems, Bilbao (Spain) 17th - 19th October 2007*, 2007.
20. S. Ranise and C. Tinelli. Satisfiability modulo theories. *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, 21(6):71–81, 2006.
21. SC-167. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1992.

22. Bastian Schlich, Falk Salewski, and Stefan Kowalewski. Applying model checking to an automotive microcontroller application. In *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)*. IEEE, 2007. ISBN 1-4244-0840-7.
23. Ofer Strichman. On solving presburger and linear arithmetic with sat. In M.D. Aagaard and J.W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD),*, number 2517 in LNCS, pages 160–170. Springer, 2002.
24. G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, page pp. 115. Consultants Bureau, New York-London, 1962.
25. Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proceedings of the PLDI'04, June 9-11, 2004, Washington, DC, USA*. ACM 1581138075/04/0006.
26. Verified Systems International GmbH, Bremen. *RT-Tester 6.2 – User Manual*, 2007.