A Unified Approach to Abstract Interpretation, Formal Verification and Testing of C/C++ Modules

Jan Peleska

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

Abstract. In this paper, a unified approach to abstract interpretation, formal verification and testing is described. The approach is applicable for verifying and testing C/C++ functions and methods and complies with the requirements of today's applicable standards for the development of safety-critical systems in the avionics and railway domains. We give an overview over the techniques required and motivate why an integrated approach is not only desirable from the verification specialists' perspective, but also from the tool builders' point of view. Tool support for our approach is available, and it is currently applied in industrial verification projects for railway control systems. All techniques can be adapted to model-based testing in a straightforward way. The objective of this article is to describe the interplay between the methods, techniques and tool components involved; we give references to more comprehensive descriptions of the underlying technical details.

1 Introduction

1.1 Overview

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[1] combining abstract interpretation, formal verification by model checking 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. From the practitioners' point of view, our approach is driven by the applicable standards for safety-critical systems development in the railway and avionic domains: The methods and techniques described should help

[1] We use the term *module* to denote both C functions and C++ methods.

to (1) fulfil the software-quality related requirements of these standards more efficiently and (2) facilitate the formal justification that these requirements have been completely fulfilled.

We present an overview of the methods required to achieve these goals for C/C++ code verification. The tasks involved can be roughly structured into six major building blocks (see Figure 1): (1) A parser front-end is required to transform the code into an intermediate model representation which is used for the analyses to follow. The intermediate model representation contains a suitably abstracted memory model which helps us to cope with the problems of aliasing, type casts and mixed arithmetic and bit operations typically present in C/C++ code. (2) Verification tasks have to be decomposed into sub-tasks investigating sub-models. A sub-model selector serves for this purpose. (3) Concrete, symbolic and abstract interpreters are required to support the process of constraint generation, the abstract interpreter serving the dual purpose of runtime error checking and of constraint simplification. (4) A constraint generator prepares the logical conditions accumulated by the interpreters for the (5) constraint solver which is needed to calculate concrete solution vectors as well as over and under approximations of the constraint solution sets. (6) For automated test case generation, test data is constructed as solutions to the constraints associated with a specific reachability goal. The test data has to be integrated in test procedures automatically invoking the tested modules, feeding the input data to their interfaces and checking the modules' behaviour against expected results specifications. Test procedures are internally represented as abstract syntax trees, so that different syntax requirements of test execution environments can be conveniently met.

Our presentation focuses on the interplay between these building blocks and provides references to more detailed elaborations of the technical problems involved.

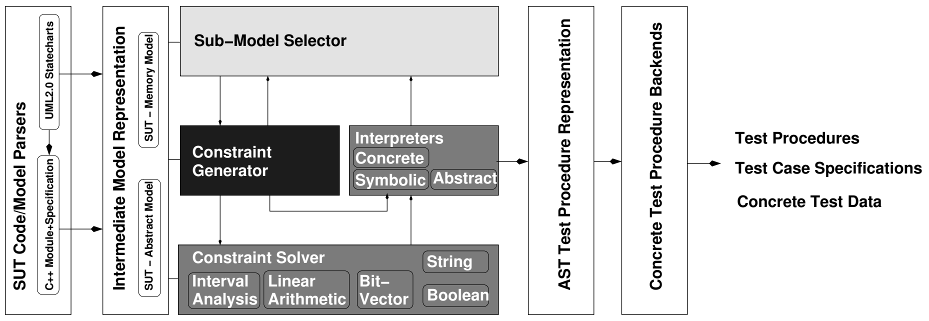

Fig. 1. Building blocks of test automation, static analysis and property verification tool platform.

In section 2 the requirements of standards related to safety-critical systems development are sketched. Section 3 contains the main part of this paper. It describes the work flow between the tool components listed above which conforms to these standards. Moreover, the methods used to implement the component functionality are sketched. Section 4 presents a conclusion.

1.2 Related Work

The work presented here summarises and illustrates results previously published by the author and his research team in cooperation with Verified Systems International GmbH [3, 17, 19, 16, 15].

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 [11] 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 [10] 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 [14]. In contrast to [11], 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 [6, 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 [8], 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 [6, 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 solver implemented in the tool relies on ideas developed in [9] 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 [12].

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

According to the standards [21, 7, 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 [13, 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.

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.

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

3.1 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. 2). Optionally, additional assertions can be inserted into an "inspection copy" of the module code. The *Unit Under Test (UUT)*[2] 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. 2, 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

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

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.

```
 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. 2. Example: Module specification by pre- and post-conditions.

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) 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.

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 code sections in the UUT can be triggered in a simple way.

Complementary to functional testing, it is required to perform *structural testing*. The goal of structural testing consists in covering the UUT control structures, statements, calls to sub-functions and interfaces, while still checking that the functional requirements are met. Currently, we support the coverage criteria required in the standards [21, 7]:

– 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 $\mathtt{f}_{\mathtt{aug}}()$ shown in Fig. 3. Obviously, statement coverage of $\mathtt{f}_{\mathtt{aug}}()$ implies functional coverage of `f()` in the sense exemplified above.

```
void f_aug(t1 x1, ..., tn xn) {
  t r;
  if ( P(v) ) {
    // This branch is entered when input data
    // satisfied pre-condition P(v)

    v0 = v;            // Create copy of pre-states
    r = f(x1, ...,xn); // Call the UUT

    // Post-state has changed variable vector v,
    // pre-state is saved in auxiliary variable v0.

    if ( C_1(v0,v) ) {
      assert( Q_1(v0,v) );
    }
    ...
    if ( C_k(v0,v) ) {
      assert( Q_k(v0,v) );
    }
  }
}

```

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

For the abstract interpretation 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.

3.2 Transformation into an Intermediate Model Representation

To facilitate the re-use of algorithms for testing and verifying programs written in other programming languages and to support model-based testing and verification approaches, all algorithms operate on an *intermediate model representation IMR*. Conceptually, IMRs consist of collections of transition systems $T = (S, S_0, \longrightarrow)$ which may be connected by a decomposition relation (e. g. transition system state $s \in S$ is decomposed into one or more sub-ordinate transition systems $T_1, \ldots, T_n$) and a parallelism relation (transition system T_1 is executed in parallel to T_2).

Since we do not impose any restrictions on the size of the data types involved, explicit transition system state space representations of C/C++ modules in the IMR would be impossible. Instead, the IMR encodes the transition relation, using a combined explicit and symbolic technique: The full transition system state space S is structured into *locations Loc* and variable valuations $V \not\rightarrow D$, i. e., $S = \mathit{Loc} \times (V \not\rightarrow D)$, where V denotes the set of symbols and D a suitable domain capturing all symbol types involved. Note that the valuation mappings are partial, because at different states different symbols may be present in the state-dependent scope. Moreover, V may be infinite to allow for symbols specified by de-referenced pointer expressions (such as `*(p->next->...->next->x)`) or array elements with arbitrary index expressions (like $\mathtt{a}[\mathtt{i}_0 + \ldots + \mathtt{i}_\mathtt{n}]$).

A directed *location graph* $L = (\mathit{Loc}, \longrightarrow_L \subseteq \mathit{Loc} \times \mathit{Label} \times \mathit{Loc})$ with labelled edges explicitly represents an abstraction of the transition system. The abstraction hides all concrete symbol valuations. The Edges $e = l_0 \longrightarrow_L l_1$ of L may be labelled by *guard conditions* $g(e)$, that is, predicates with symbols from V as free variables. The guard conditions specify the constraints on variables valuations to be fulfilled for having an associated transition in the concrete transition system $T = (\mathit{Loc} \times (V \not\rightarrow D), S_0, \longrightarrow)$. Furthermore, edges e can be annotated with symbolic transition relations specifying *actions* $\epsilon(e)$, that is, changes $\sigma_1 = \epsilon(e)(\sigma_0)$ on symbol valuations accompanying a $(l_0, \sigma_0) \longrightarrow_L (l_1, \sigma_1)$-transition in the concrete transition system T. Similarly, nodes l_1 of the location graph can be annotated by *entry actions* $\alpha(l_1)$, specifying changes on symbol valuations occurring when entering location l_1. Furthermore, they can be labelled with *invariants inv*(l) and *do-actions* $\delta(l)$ to encode models specified in timed, potentially hybrid, formalisms and supporting urgent and non-urgent transitions.

A pre-requisite for a concrete transition $(l_0, \sigma_0) \longrightarrow (l_1, \sigma_1)$ to take place in T is that there exists an edge $l_0 \xrightarrow{[g]/a}_L l_1$ in the location graph such that $\sigma_0 \models g$, that is, $g(\sigma_0(x_0)/x_0, \ldots, \sigma_0(x_n)/x_n)$ evaluates to `true`. This is obviously independent on the concrete formalism encoded in the IMR. The more specific rules for deriving possible T-transitions depend on the underlying formalism. As a consequence, we instantiate specific interpreters implementing the concrete transition rules with each supported formalism. This necessity suggests an object-oriented approach for the IMR.

For C/C++ module testing, each module $f()$ corresponds to one transition system $T(f)$ and transition system states correspond to computation states of the module. A call from $f()$ to a sub-module $h()$ corresponds to a state s representing the call which is related to a sub-ordinate transition system $T(h)$. The IMR uses GIMPLE control flow graphs (CFG) as location graphs for C/C++ modules (see [10]). These graphs have one dedicated *entry node* `BLOCK 0` and one *exit node* `EXIT`. Each location is associated with an entry action, and these are the only actions defined for CFGs; do-actions, invariants and actions associated with edges are not needed. Actions are defined in imperative programming language style according to the GIMPLE syntax and in 3-address code[3]. Each CFG node l has at most two outgoing edges $l \xrightarrow{[g_0]}_L l', l \xrightarrow{[g_1]}_L l''$ corresponding to if-else-conditions, so $g_1 = \neg g_0$. The symbol set V consists of the variable symbols occurring in $f()$ plus additional atomic variables introduced to support the 3-address code representation. Each concrete transition of T can be derived from the rule

$$\frac{l_0 \xrightarrow{[g]}_L l_1,\ \sigma_0 \models g}{(l_0, \sigma_0) \longrightarrow (l_1, \alpha(l_1)(\sigma_0))}$$

A *run* of a C/C++ module is a finite computation, that is, a sequence

$$r = \langle (l_0, \sigma_0), \ldots, (l_n, \sigma_n) \rangle$$

such that $(l_0, \sigma_0) \in S_0$ and

$$\forall\ i \in \{0, \ldots, n-1\} : \exists\ l_i \xrightarrow{[g_i]}_L l_{i+1} : \sigma_i \models g_i \wedge \sigma_{i+1} = \alpha(l_{i+1})(\sigma_i)$$

A path $l_0 \longrightarrow l_1 \longrightarrow, \ldots, \longrightarrow l_n$ through the location graph $L(T)$ is called *feasible* if an associated run in T can be constructed, otherwise the path is *infeasible*.

If the entry action of the target node consists of a function call then the following rule for the calculation of $\alpha(l_1)(\sigma_0)$ is applied:

$$\frac{\alpha(l_1) = \{\mathtt{x_0 = h(x_1, \ldots, x_n);}\},\ (BLOCK\ 0, \sigma_0|_h) \longrightarrow_h^* (EXIT, \sigma_1)}{\alpha(l_1)(\sigma_0) = (\sigma_1|_f)[x_0 \mapsto \sigma_1(h_{return})]}$$

This rule is interpreted as follows: If $T(f)$ may perform a transition into location l_1 which has a function call as entry action, then the effect of this action is defined by $T(h)$. If $T(h)$ transforms entry valuation $\sigma_0|_h$ into exit valuation σ_1 then the symbols still visible at the level of $f()$ (that is, everything but the formal parameters and stack variables of $h()$) carry the new valuation σ_1, and the return value of $h()$ is assigned to the target variable x_0 of the assignment. The symbol $|$ in $\sigma_0|_h$ denotes (1) the extension of dom σ_0 to the scope of $h()$: dom σ_0 is now extended by the formal parameters and stack variables of $h()$. (2) The assignment of actual parameter values used in the call to h to formal parameter valuations visible inside h. Observe that for reference parameters the formal parameter gets an address assignment from the associated actual parameter. Conversely, $\sigma_1|_f$ denotes (1) the domain restriction of valuation function σ_1; formal parameters and local variables of $h()$ are no longer visible, and (2) the assignment of the return value of $h()$ to an intermediate variable h_{return} visible at the level of $f()$.

[3] Exceptions are calls to modules with more than 3 parameters $\mathtt{y = f(x_1, \ldots, x_n)}$ and access to multi-dimensional array $\mathtt{y = a[x_1] \ldots [x_n]}, n > 2$.

Due to the aliasing effects possible in C/C++, the sub-function $h()$ may indirectly change local variables of $f()$ via assignments to de-referenced pointers. As a consequence, the effect of the $h()$-execution on symbol valuations "apparently" outside the scope of $h()$ can be quite complex. The memory model and the associated valuation rules described below have been designed to cope with these problems. For the moment it suffices to observe that an assignment to a symbol inside the scope of $h()$ may implicitly change the valuation of (due to recursive data structures and pointer de-referencing) possibly infinitely many other symbols which may even be outside the scope of $h()$.

3.3 The Sub-Model Generator

The reason for using a mixed explicit (location graph) and symbolic (specification of transition effects on valuations) intermediate model representation lies in the fact that this allows us to distribute the elaboration of reachability strategies onto two tool components – the solver and the sub-model generator – instead of only one (the solver). It has been pointed out in [3] that the reachability goals associated with structural test coverage and with the verification of safety properties can always be expressed as a goal to cover specific edges in a location graph; for C/C++ this is the GIMPLE CFG or a semantically equivalent transformation thereof. The task of the sub-model generator is therefore to restrict the complete transition system collection representing the UUT into a collection of restricted sub-systems by eliminating as many original transitions that will never be visited by any path leading to the destination edges as possible. Since this should be performed in an efficient manner *before* a constraint solver is involved, the sub-model generator performs a conservative approximation based on the location graph alone, that is, without calculating symbol valuations. Furthermore, the sub-model generator receives feed-back from the constraint solver about infeasible paths through the location graph and applies learning strategies for avoiding to pass infeasible sub-models to the solver. Finally, this tool component keeps track of the location graph coverage achieved.

The simplest sub-models are paths through the location graph, more complex ones are

– trees leaving a higher degree of freedom for the solver in order to construct runs to the destination edges, and
– sub-graphs representing if-else branches both leading back to the same path to the destination edge.

Example 1. Consider a C/C++-UUT whose transition relation is encoded by the CFG depicted in Fig. 4. For structural testing it is useful to know all paths up to a certain depth leading to any given edge. For this purpose, the sub-model generator maintains a tree of depth k as depicted in Fig. 5, associated with a function ϕ_k mapping edges e of the location graph to lists of nodes n in the tree, such that a path in the tree from root to n corresponds to a path trough the transition graph reaching e. For the configuration described by Fig. 4 and

5 we have, for example, $\phi_6(f) = \langle (l_5, 3), (l_5, 5), (l_5, 4), (l_5, 7) \rangle$. If one path, say, the one specified by $(l_5, 3)$, to the destination edge is identified by the solver to be infeasible, the tree is pruned at the target node of the destination edge. In our example, edges in the sub-tree starting at $(l_5, 3)$ would never be suggested again by the sub-model generator. If all paths specified by $\phi_k(f)$ turned out to be infeasible, the tree can be expanded if possible, but only at leaves which do not reside in sub-trees already pruned.

For structural testing it will be too costly to expand the tree of Fig. 5 further, if most of the edges have already been covered. The sub-model generator now constructs another tree structure capturing all (still potentially feasible) paths to an edge still uncovered.

More details about the algorithms for generating sub-models can be found in [3]. □

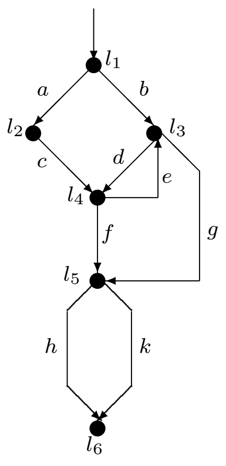

Fig. 4. Location graph example.

3.4 Interpreters

Symbolic Interpretation. Given the IMR of a spcification or C/C++ module, the symbolic interpreter performs symbolic computation of runs through a sub-model of the location graph. This interpreter is the core component for generating the constraints to be solved for the inputs of a module, in order to reach a given edge.

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

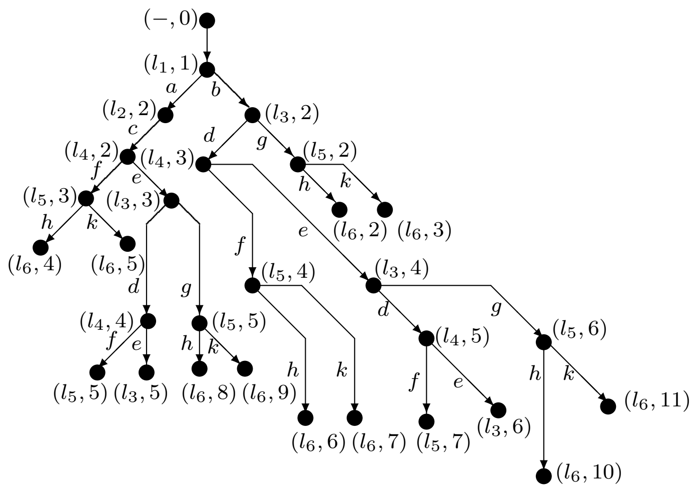

Fig. 5. Tree sub-model with paths to *all* edges in the location graph.

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. 6). 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.

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 2. Suppose that variables `float` $\mathtt{x}, \mathtt{y}, \mathtt{z}$; 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

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

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

$m.v_0$ 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. 6. Structure of a memory item m.

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 $\&x + 31$. 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 $y_n + z_n$. $\square$

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

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

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

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

Example 3. 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

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

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

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

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

so a transition can be performed on symbolic level whenever a corresponding edge exists in the control flow graph ($\overset{g}{\longrightarrow}_{CFG}$ denotes the edge-relation in the module's CFG, with guard condition g as label). It may turn out, however, on abstract or concrete interpretation level, that such a transition is infeasible because 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' 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.

In [16, 15], formal transition rules have been specified for $\longrightarrow_G$, as well as the algorithms required for rule application. Here we will only present an example, in order to illustrate the effect of these rules on the symbolic memory state.

Example 4. 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:

$$
\begin{aligned}
m_p^1 &= (1, 3, \&\texttt{a[0]}, 32 \cdot p, 32 \cdot p + 32, \texttt{int}, \texttt{Undef}, 0 \le p \wedge p < 10)\\
m^2 &= (2, \infty, \&\texttt{i\_0}, 0, 32, \texttt{int}, \texttt{i}_1, \texttt{true})\\
m^3 &= (3, \infty, \&\texttt{D\_4151}, 0, 32, \texttt{int}, \texttt{m}_2 + \texttt{n}_2\texttt{true})\\
m_p^4 &= (4, 5, \&\texttt{a[0]}, 32 \cdot p, 32 \cdot p + 32, \texttt{int}, \texttt{Undef}, 0 \le p \wedge p < 10 \wedge p \neq i\_0_2)\\
m^5 &= (4, 5, \&\texttt{a[0]}, 32 \cdot i\_0_2, 32 \cdot i\_0_2 + 32, \texttt{int}, D\_4151_3, 0 \le i\_0_2 \wedge i\_0_2 < 10)\\
m^6 &= (5, \infty, \&\texttt{j\_1}, 0, 32, \texttt{int}, \texttt{j}_4, \texttt{true})\\
m_p^7 &= (6, \infty, \&\texttt{a[0]}, 32 \cdot p, 32 \cdot p + 32, \texttt{int}, \texttt{Undef}, 0 \le p \wedge p < 10 \wedge p \neq i\_0_2 \wedge p \neq j\_1_5)\\
m^8 &= (6, \infty, \&\texttt{a[0]}, 32 \cdot i\_0_2, 32 \cdot i\_0_2 + 32, \texttt{int}, D\_4151_3,\\
&\qquad 0 \le i\_0_2 \wedge i\_0_2 < 10 \wedge i\_0_2 \neq j\_1_5)\\
m^9 &= (6, \infty, \&\texttt{a[0]}, 32 \cdot j\_1_5, 32 \cdot j\_1_5 + 32, \texttt{int}, 0, 0 \le j\_1_5 \wedge j\_1_5 < 10)
\end{aligned}
$$

Initially, the declared array `a` is undefined because it resides in the stack segment where no variable initialisation takes place (memory item m_p^1). The assignment to `a[i_0]` in line 4 invalidates the original item m_p^1 representing the symbolic valuation of `a`, so $m_p^1.v_1 = 3$. This leads to the creation of two new items: m^5 specifies the effect of the assignment in line 4, and m_p^4 specifies the array elements which are still undefined. A further invalidation of m_p^4, m^5 is caused by the assignment in line 6 and generates the new items m_p^7, m^8, m^9. Item m^8, for example, specifies the situation where the original value written to `a[i_0]` in line 4 is still visible after the new assignment in line 6. □

Abstract Interpretation. 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.

Concrete Interpretation. The concrete interpreter applies concrete GIMPLE semantics [16] in order to find out the paths through the IMR that are covered with concrete sets of input data. It is applied

– in verification to present counter examples,

– in structural testing to determine the location graph edges following a reachable destination edge which are also covered before the exit point of a module execution is reached.

3.5 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 5. Let us extend Example 4 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 $\Phi := \mathtt{D\_4160} < 0$.

(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) $\mathtt{a}_7[\mathtt{i\_0}_7]$ matches with items m^7_p, m^8, m^9 for a and m^2 for i_0 in Example 4, since the other items with base address &a[0] are already outdated at computation step 7; this leads to resolutions

$$\begin{aligned}\Phi = {} & \mathtt{D\_4160} < 0 \wedge ((\mathtt{D\_4160} = \mathtt{Undef} \wedge \mathtt{i\_0}_7 = \mathtt{p} \wedge 0 \le \mathtt{p} \wedge \mathtt{p} < 10 \wedge \mathtt{p} \neq \mathtt{i\_0}_2 \wedge \mathtt{p} \neq \mathtt{j\_1}_5) \vee \\ & (\mathtt{D\_4160} = \mathtt{D\_4151}_3 \wedge \mathtt{i\_0}_7 = \mathtt{i\_0}_2 \wedge 0 \le \mathtt{i\_0}_2 \wedge \mathtt{i\_0}_2 < 10 \wedge \mathtt{i\_0}_2 \neq \mathtt{j\_1}_5) \vee \\ & (\mathtt{D\_4160} = 0 \wedge \mathtt{i\_0}_7 = \mathtt{j\_1}_5 \wedge 0 \le \mathtt{j\_1}_5 \wedge \mathtt{j\_1}_5 < 10)) \wedge \\ & \mathtt{i\_0}_7 = \mathtt{i\_0}_2 \wedge \mathtt{i\_0}_2 = \mathtt{i}_1 \wedge \mathtt{j\_0}_5 = \mathtt{j}_4 \wedge \mathtt{j}_4 = \mathtt{j}_1\end{aligned}$$

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 $\mathtt{i\_0}_7, \mathtt{i\_0}_2, \mathtt{j\_1}_5, \ldots$), 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 $32 \cdot \mathtt{i\_0}_7 = 32 \cdot \mathtt{j\_1}_5$. 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) Simplify by means of abstract interpretation: Using interval analysis for symbols of numeric type, some atoms of the constraint can be quickly verified or falsified, in order to simplify the constraint finally passed to the solver. Suppose,

for example, that `i, j` were inputs to the module and fulfilled the pre-conditions $0 \leq i < 2$, $2 \leq j < 10$ The interval analysis would yield `true` for condition $i\_0_2 \neq j\_1_5$ for all elements i, j satisfying the pre-condition, so conjunct $i\_0_2 \neq j\_1_5$ could be deleted in Φ.

(5) Prepare the constraint for the solver: Following the restrictions for admissible constraints described in [9], our solver requires some pre-processing of Φ: (a) Inequalities like $\mathtt{i\_0}_2 \neq \mathtt{j\_1}_5$ are replaced by disjunctions involving $<, >$, e. g. $\mathtt{i\_0}_2 < \mathtt{j\_1}_5 \vee \mathtt{i\_0}_2 > \mathtt{j\_1}_5$. (b) Inequalities $a < b$ are only admissible if a or b is a constant. Therefore atoms like $\mathtt{i\_0}_2 < \mathtt{j\_1}_5$ 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 $\mathtt{i\_0}_2 + \mathtt{s} = \mathtt{j\_1}_5 \wedge \mathtt{0} < \mathtt{s}$. (c) Three-address-code is enforced, so that – with the exception of function calls $\mathtt{y} = \mathtt{f}(\mathtt{x}_0, \ldots, \mathtt{x}_\mathtt{n})$ and array expressions $\mathtt{y} = \mathtt{a}[\mathtt{x}_1] \ldots [\mathtt{x}_\mathtt{n}]$ – 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.

3.6 Constraint Solver

The solver handling the 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* [12] and learning strategies designed by [9], see also [3] for more details of solver application in the context of test automation.

4 Conclusion

We have described an integrated approach for automated testing, static analysis by abstract interpretation and formal verification by model checking (reachability analysis for safety properties). The main focus of the presentation was on the verification of C/C++ modules. It has been indicated, however, how more abstract specification models can be encoded in the same intermediate model representation IMR used for code verification. As a consequence, the algorithms operating on the IMR can be directly applied to model-based testing and model verification. The techniques described in this paper, together with the tool support provided by the test automation system RT-Tester [26] are applied in industrial projects in the fields of railway control systems and avionics, the model-based approach is currently applied in the railway and automotive domains. More details about model-based testing can be found in [18].

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. Miné, 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. 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.
7. European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems.* CENELEC, Brussels, 2001.
8. 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.
9. 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.
10. GCC, the GNU Compiler Collection. The GIMPLE family of intermediate representations. See `http://gcc.gnu.org/wiki/GIMPLE`.
11. 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.
12. Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis.* Springer-Verlag, London, 2001.
13. Nancy G. Leveson. *Safeware.* Addison-Wesley, 1995.

14. Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen, May 2007.
15. Jan Peleska. Integrated and automated abstract interpretation, verification and testing of C/C++ modules. In Dennis R. Dams, Ulrich Hannemann, and Martin Steffen, editors, *Correctness, Concurrency and Compositionality – Festschrift for Willem-Paul de Roever*, LNCS Festschrift series. Springer, 2008. To appear.
16. 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.
17. 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.
18. Jan Peleska, Oliver Möller, and Helge Löding. Model-based testing for model-driven development with uml/dsl. In *To appear in Proceedings of the Software & Systems Quality Conference 2008 (SQC 2008)*. Available under `http://www.informatik.uni-bremen.de/agbs/jp/jp_papers_e.html`.
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.