Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 217 (2008) 113–131
www.elsevier.com/locate/entcs

Symbolic and Abstract Interpretation for C/C++ Programs

Helge Löding[2,4]

GESy Graduate School of Embedded Systems
University of Bremen
and Verified Systems International GmbH
Germany

Jan Peleska[1,3]

Centre of Information Technology
University of Bremen
Germany

Abstract

We present a construction technique for abstract interpretations which is generic in the choice of data abstractions. The technique is specialised on C/C++ code, internally represented by the GIMPLE control flow graph as generated by the gcc compiler. The generic interpreter handles program transitions in a symbolic way, while recording a history of symbolic memory valuations. An abstract interpreter is instantiated by selecting appropriate lattices for the data types under consideration. This selection induces an instance of the generic transition relation. All resulting abstract interpretations can handle pointer arithmetic, type casts, unions and the aliasing problems involved. It is illustrated how switching between abstractions can improve the efficiency of the verification process. The concepts described in this paper are implemented in the test automation and static analysis tool RT-Tester which is used for the verification of embedded systems in the fields of avionics, railways and automotive control.

Keywords: automated testing, static analysis, abstract interpretation, Galois connections

1 Introduction

1.1 Objectives and Overview

Concrete and abstract interpretation are core mechanisms for automated static analysis, test case/test data generation and property checking of software: The

[1] Email: jp@tzi.de
[2] Email: hloeding@tzi.de
[3] Partially supported by the BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015B
[4] Supported by a research grant of the Graduate School in Embedded Systems GESy http://www.gesy.info

doi:10.1016/j.entcs.2008.06.045

concrete interpretation helps to explore program (component) behaviour with concrete data values without having to compile, link and execute the program on the target platform. The abstract interpretation reduces the complexity of verification goals or, more general, reachability problems, by abstracting from details which are unnecessary for the goal under consideration.

Consider the building blocks typically present in tools supporting test automation, static analysis and/or property checking as shown in Fig. 1: The program code to be analysed or a specification model are transformed into a uniform *intermediate model representation (IMR)* which is independent of the concrete SUT code or specification syntax. This reduces the dependencies between concrete syntax and analysis algorithms. Most of the problems arising in automated test case/test data generation, static analysis and property verification can be paraphrased as reachability problems, as has been pointed out in [10]. Therefore a *path selector* performs a choice of potential paths through the model to be checked with respect to feasibility: The goal is solved if concrete input data can be found so that the software component under analysis executes along one of the suggested paths. While the general reachability problem is undecidable, concrete goals can often be realised in a highly efficient way. To this end, the *constraint generator* constructs a collection of constraints to be met in order to provoke an execution along the selected paths. The construction requires a *symbolic interpreter*, a tool component for collecting the guard conditions along the selected paths. With a sufficient collection of constraints at hand, the *constraint solver* tries to construct concrete data solving the constraints or to prove their infeasibility.

The choice of the *abstract* interpretation technique considerably influences the efficiency of automated solvers used for these purposes: For proving that a constraint collection can never be satisfied it is often more efficient to show this for an abstracted program version, so that this also implies infeasibility for the concrete program. Conversely, some abstractions are especially useful for under-approximating the solution set of the constraints given, so that any data vector of this approximation represents a solution.

In this paper we focus on interpreters for C/C++ programs. For this task it is necessary to capture all "side effects" of aliasing, pointer arithmetic, type casts and unions possibly occurring in C/C++ software, so that no hidden effects of instructions on the valuation of symbols not occurring in the statement are missed during the interpretation process. We first present operational rules for a concrete semantics covering these aspects (Section 3). Next we observe that for a given collection of constraints, the efficiency of the solver strongly depends on the choice of abstraction. As a consequence it is desirable to switch abstractions for one and the same data type during the interpretation while still ensuring the correctness of the interpretation results. This objective is met by means of a *symbolic* interpreter for C/C++ programs (Section 4): This tool component handles program transitions in a symbolic way, while recording a history of symbolic memory valuations. The valuations are represented by memory addresses (these are necessary in order to cope with the aliasing problems), value expressions and application conditions: A

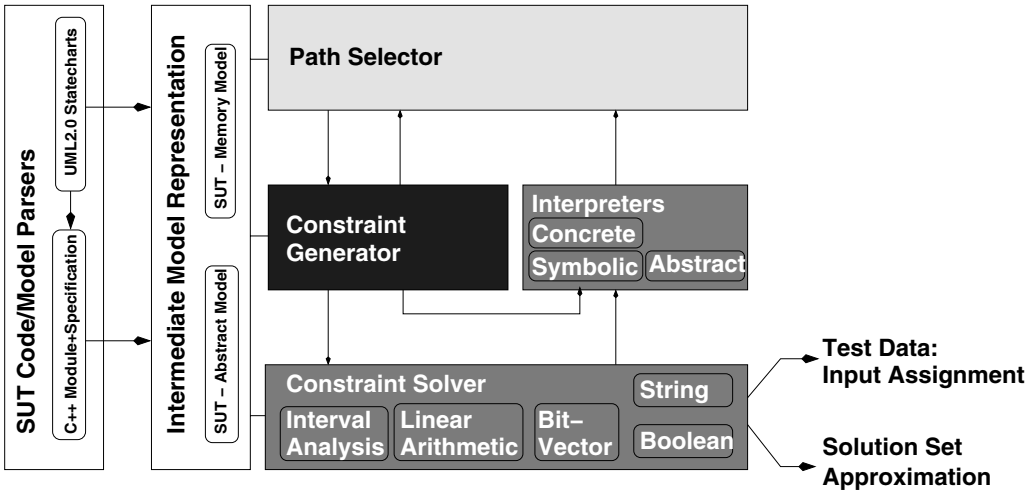

Fig. 1. Building blocks of tools for test automation, static analysis and property verification.

memory item is only valid if a valuation of inputs can be found so that the application condition becomes true. Finally we describe how abstract interpreters can be constructed by instantiating the symbolic interpreter with lattices to be used for abstracting the data types involved (Section 5). As a consequence, the basic interpretation algorithm can be completely re-used for each choice of abstraction lattice, only functions for the valuation of expressions in the context of the selected lattices have to be added. In Section 6 an example is given which illustrates the mechanics and the effects of symbolic and abstract interpretation.

1.2 Background and Related Work

The full consideration of C/C++ aliasing situations with pointers, casts and unions is achieved at the price of lesser performance. In [4,2], for example, it is pointed out how more restrictive programming styles, particularly 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 [14] 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 [5], 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 [12]. 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 authors and their research group at the University of Bremen in cooperation with Verified Systems International GmbH [15]. In [10] we have motivated in more detail why testing, static analysis and property checking of software code should be considered as an integrated verification task, so integrated tool support for these complementary aspects of software verification is highly desirable. The approach pursued with the RT-Tester tool differs from the strategies of other authors [4,2,14]: We advocate an approach where test and 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 and interactive support for bug-localisation and further investigation of potential failures is provided. The SMT constraint solver used in the tool is based on ideas described in [11,1,6].

2 Theoretical Foundations

Recall that a binary relation $\sqsubseteq$ on a set L is called a *(partial) order* if $\sqsubseteq$ is reflexive, transitive and anti-symmetric. An element $y \in L$ is called an *upper bound of* $X \subseteq L$ if $x \sqsubseteq y$ holds for all $x \in X$. The lower bound of a set is defined dually. An upper bound y' of X is called a *least upper bound of* X and denoted by $\sqcup X$ if $y' \sqsubseteq y$ holds for all upper bounds y of X. Dually, the *greatest lower bound* $\sqcap X$ of a set X is defined.

An ordered set $(L, \sqsubseteq)$ is called a *complete lattice*, if $\sqcap X$ and $\sqcup X$ exist for all subsets $X \subseteq L$. Lattice L has a *largest element* (or *top*) denoted by $\top =_{\text{def}} \sqcup L$ and a *smallest element* (or *bottom*) denoted by $\bot =_{\text{def}} \sqcap L$. Least upper bounds and greatest lower bounds induce binary operations $\sqcup, \sqcap : L \times L \to L$ by defining $x \sqcup y =_{\text{def}} \sqcup\{x, y\}$ (the *join* of x and y) and $x \sqcap y =_{\text{def}} \sqcap\{x, y\}$ (the *meet* of x and y), respectively. If the join and meet are well-defined for an ordered set $(L, \sqsubseteq)$ but $\sqcup X, \sqcap X$ do not exist for all $X \subseteq L$ then $(L, \sqsubseteq)$ is called an *(incomplete) lattice*.

From the collection of canonic ways to construct new lattices from existing ones $(L, \sqsubseteq), (L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2)$, we need (1) cross products $(L_1 \times L_2, \sqsubseteq')$ where the partial order is defined by $(x_1, x_2) \sqsubseteq' (y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$ and (2) partial function spaces $(V \not\to L, \sqsubseteq')$ where $f \sqsubseteq' g$ for $f, g \in V \not\to L$ if and only if $\text{dom } f \subseteq \text{dom } g \wedge (\forall x \in \text{dom } f : f(x) \sqsubseteq g(x))$.

Mappings $\phi : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ between ordered sets are called *monotone* if $x \sqsubseteq_1 y$ implies $\phi(x) \sqsubseteq_2 \phi(y)$ for all $x, y \in L$. Mappings $\phi : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ between lattices are called *homomorphisms* if they respect meets and joins, that is, $\phi(x \sqcup_1 y) = \phi(x) \sqcup_2 \phi(y)$ and $\phi(x \sqcap_1 y) = \phi(x) \sqcap_2 \phi(y)$ for all $x, y \in (L_1, \sqsubseteq_1)$. Since $x \sqsubseteq_1 y$ implies $x \sqcup_1 y = y$ and $x \sqcap_1 y = x$, homomorphisms are monotone.

A *Galois connection (GC)* between lattices $(L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2)$ is a tuple of mappings $\_^{\triangleright} : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ (called *right*) and $\_^{\triangleleft} : (L_2, \sqsubseteq_2) \to (L_1, \sqsubseteq_1)$ (called

left) such that $a^{\triangleright} \sqsubseteq_2 b \Leftrightarrow a \sqsubseteq_1 b^{\triangleleft}$ for all $a \in L_1, b \in L_2$. This defining property implies that Galois connections are monotone in both directions.

Given any transition system $TS = (S, S_0, \longrightarrow)$ with state space S, initial states in $S_0 \subseteq S$ and transition relation $\longrightarrow \subseteq S \times S$, the most fine-grained state space abstraction possible is represented by the power set lattice $L_{\mathbf{P}}(S) = (\mathbf{P}(S), \subseteq)$ with join operation $\cup$ and meet $\cap$. We introduce an abstract interpretation semantics on $L_{\mathbf{P}}(S)$ by turning it into a state transition system $TS_{\mathbf{P}} = (L_{\mathbf{P}}(S), \{S_0\}, \longrightarrow_{\mathbf{P}})$ by lifting the original transition relation to sets: Using Plotkin-style notation, this can be specified as

$$\frac{\forall i \in I, s_i, s_i' \in S : s_i \longrightarrow s_i'}{\{s_i \mid i \in I\} \longrightarrow_{\mathbf{P}} \{s_i' \mid i \in I\}}$$

Compared to the original transition system TS, this abstract interpretation $\longrightarrow_{\mathbf{P}}$ introduces no loss of information, since its restriction to pairs of singleton sets is equivalent to the original transition relation:

$$\forall s_1, s_2 \in S : s_1 \longrightarrow s_2 \Leftrightarrow \{s_1\} \longrightarrow_{\mathbf{P}} \{s_2\}$$

It is, however, an abstraction, since for transitions between states with cardinality higher than one, say $\{s_1, s_2, \ldots\} \longrightarrow_{\mathbf{P}} \{s_1', s_2', \ldots\}$, only the possible resulting states are listed $(s_1', s_2', \ldots)$ but the information whether, for example, $s_1 \longrightarrow s_1'$ or $s_1 \longrightarrow s_2'$ is no longer available.

Now, given any other transition system $TS_L = (L, L_0, \longrightarrow_L)$ based on a lattice $(L, \sqsubseteq)$ we can check whether TS_L is a valid abstract interpretation of TS by the aid of $TS_{\mathbf{P}}$ and Galois connections:

Definition 2.1 Transition system $TS_L = (L, L_0, \longrightarrow_L)$, based on a lattice $(L, \sqsubseteq)$, is a *valid abstract interpretation of* $TS = (S, S_0, \longrightarrow)$ if (i) there exists a Galois connection $(\mathbf{P}(S), \subseteq) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} (L, \sqsubseteq)$, (ii) the transition relation $\longrightarrow_L$ is a *valid abstract relation* the sense that $\forall a, a', b \in L : (a \longrightarrow_L a' \wedge b \sqsubseteq a \Rightarrow \exists b' \in L : b \longrightarrow_L b' \wedge b' \sqsubseteq a')$, (iii) the transition relation $\longrightarrow_L$ satisfies $\forall (p, p') \in \longrightarrow_{\mathbf{P}}: \exists a' \in L : p^{\triangleright} \longrightarrow_L a' \wedge p'^{\triangleright} \sqsubseteq a'$ and (iv) the transition relation $\longrightarrow_L$ satisfies $\forall (a, a') \in \longrightarrow_L: \exists p' \in \mathbf{P}(S) : a^{\triangleleft} \longrightarrow_{\mathbf{P}} p' \wedge p' \subseteq a'^{\triangleleft}$.

The following theorem provides a "recipe" for constructing valid abstract interpretations, as soon as a GC according to Definition 2.1, (i) has been established:

Theorem 2.2 *Given lattice* $(L, \sqsubseteq)$ *and Galois connection* $(\mathbf{P}(S), \subseteq) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} (L, \sqsubseteq)$, *define transition system* $TS_L = (L, L_0, \longrightarrow_L)$ *by (i)* $L_0 = \{S_0\}^{\triangleright}$, *(ii)* $\frac{p^{\triangleright\triangleleft} \longrightarrow_{\mathbf{P}} p'}{p^{\triangleright} \longrightarrow_L p'^{\triangleright}}$ *and (iii)* $\frac{a^{\triangleleft} \longrightarrow_{\mathbf{P}} p'}{a \longrightarrow_L p'^{\triangleright}}$ *Then* TS_L *is a valid abstract interpretation of* TS *in the sense of Definition 2.1.*

For more details about lattices and GC and the proof of Theorem 2.2 the reader is referred to [3,9].

3 Control Flow Graphs and GIMPLE, Concrete Semantics

3.1 GIMPLE Programs

We use the gcc compiler to transform a given C/C++ program into GIMPLE code. As described in [7,8], this semantically equivalent representation of a program constitutes an intermediate transformation result from source to assembler, where all expressions appearing in statements contain at most one operator and (with the exception of function invocations) at most two operands. Operands may only be variable names or nested structure and array accesses (henceforth called selectors) as well as constant values. By introducing auxiliary variables, all original statements will be transformed to adhere to this requirement. Statements may therefore only be assignments from expressions to variables (or atomic selectors in the above sense). Casting and referencing/dereferencing of variables (or selectors) form expressions in themselves, and may therefore not be used as operands, but instead need to be executed as separate assignments to auxiliary variables. GIMPLE programs contain no loop constructs. Instead, all loops from the original source are transformed into conditional jumps to preceeding labelled statements. GIMPLE therefore contains only two different types of branching statements:

```
<if-else-stmt> ::= if ( <condition> ) goto <label>; else goto <label>;
<switch-stmt>  ::= switch ( <variable> ) { <cases>  <default>_opt }
<cases>        ::= <cases>_opt case <value>: goto <label>;
<default>      ::= default: goto <label>;
```

For the description of concrete GIMPLE semantics we encode each GIMPLE function as a *control flow graphs (CFGs)*. Each function/method of a C/C++ program is associated with a CFG. Each CFG G has a distinguished initial node $I(G)$ corresponding to function entry and a terminal node $O(G)$ corresponding to function return. Each CFG node is labelled with a single GIMPLE statement, each edge with a GIMPLE branching condition. For sequences of non-branching statements, the edges are labelled with `true`. Branching statements are represented as edges labelled with the applicable branching conditions, each edge pointing to the target node referenced in the `goto <label>` statement in the GIMPLE code.

The concrete operational semantics of a GIMPLE program P, represented by a collection of control flow graphs as described above, will now be explained by associating a transition system with P.

3.2 GIMPLE state space

For representing the semantics of GIMPLE programs P, we use the following class of transition systems $TS_G = (S_G, S_0, \longrightarrow_G)$. The program state space is defined as

$$S_G = N(P) \times (Seg \times \mathbb{N}_0 \nrightarrow Symbols) \times (Seg \times \mathbb{N}_0 \nrightarrow BYTE^*)$$

with typical element $(n, \nu, \mu) \in S_G$. Set $N(P)$ comprises all nodes in the CFGs associated with any function of P. The second and third component of this Cartesian product represent function spaces for address mappings and memory state: For modelling the association between variables, their aliases and their associated mem-

ory portions, we introduce (1) a partial function $\nu : Seg \times \mathbb{N}_0 \nrightarrow Symbols$ mapping existing virtual addresses on the segment of type $Seg = \{stack, heap, global, code\}$ to a symbol (variable or function) associated with this address and (2) a partial function $\mu : Seg \times \mathbb{N}_0 \nrightarrow BYTE^*$ associating with each existing virtual address a sequence of bytes, representing the current memory valuation of the given address.

The set $Symbols$ only contains the *basic* symbol names, that is, the name `a` of an array, but not the array element `a[4]` and the name of a structured variable `x` but not the name of `x.y.z[5]` of a structure component. Component and array element identifiers are called *selectors* and comprised in a (possibly infinite) set $Selectors$ which is a superset of $Symbols$, since each basic name is a selector, too.

The initial state of S_G is $S_0 = \{(I(f), \nu_0, \mu_0)\}$, where $I(f)$ is the initial node of the CFG associated with the GIMPLE function of interest, ν_0 contains all addresses of global variables and actual parameters used in the invocation of $f()$ and μ_0 contains the memory portions associated with these actual parameters and of all global variables, initialised according to the precondition on which the execution of $f()$ should be based.

3.3 Auxiliary functions

For recording state changes in S_G and determining the current state of variable valuations some auxiliary functions are needed.

Given an arbitrary selector, function $\beta : Selectors \to Symbols$ returns its base symbol, e.g. for $\beta(\texttt{x.y.z[5]}) = x$. This will be required to retrieve base addresses for selectors by means of ν.

Since virtual addresses are unique across memory segments, a function $\hat{\nu} : Symbols \nrightarrow \mathbb{N}_0$ mapping identifiers to their respective address is well-defined when taking scoping into account. For a given symbol that is defined both within the stack and global segments, $\hat{\nu}$ will return the virtual address corresponding to the symbol definition within the stack.

$\hat{\nu}$ can be extended to map from selectors to virtual base addresses to yield $\nu^- : Selectors \nrightarrow \mathbb{N}_0$ with $\nu^-(sel) =_{\text{def}} \hat{\nu}(\beta(sel))$.

Given an arbitrary selector, function $\omega : Selectors \to \mathbb{N}_0$ returns the *bit offset* of the selector's memory location from its base address. The offset is measured in bits so that also operations on bitfields can be captured. This information is obviously platform-specific: ω is constructed from the size and alignment information provided by the gcc compiler on the specific platform it is used. As with ν^-, the appropriate memory segment for multiply defined base symbols is determined by first assessing symbol definitions within the stack segment.

Function $\tau : Selectors \nrightarrow Types$ returns the type for any given selector. The type information is then gained from the internal type data gathered by the gcc compiler. Again, scoping is taken into account.

Function τ may be extended to determine the type of a given expression forming $\tau^* : Expr \nrightarrow Types$ by taking (return) types of used operands and operators into account. If a given selector corresponds to a pointer type, then function

$\vec{\tau} : Selectors \not\to Types$ may be used to obtain its target type.

Function $\sigma : Types \to \mathbb{N}_0$ is used to determine a given type's size in bits.

The state space only records the current memory state as sequences of bytes. Function $\iota : BYTE^* \times Types \not\to D$ is used to interpret a given sequence of bytes as a specific type. Here, D denotes the union of all atomic domains. It is only defined for byte sequences long enough to hold a value of given type. Conversely, we define $\iota^- : D \times Types \not\to BYTE^*$ to be the byte representation for a given value with known type. For these functions, the size of atomic types, encoding methods and the little or big endianess of the platform has to be determined. This information is retrieved from the gcc type- and debugging information.

For reading data from memory, we initially define $\epsilon^a : S_G \times \mathbb{N}_0 \times \mathbb{N}_0 \not\to BYTE^*$. Function application $\epsilon^a((\nu, \mu), a, s)$ reads a bit sequence of a given length s beginning from a specified address a within the memory, and returns its contents as byte sequence. For this, we find the segment and base address (seg, a_{base}) within $dom(\nu)$, for which byte sequence $\mu(seg, a_{base})$ encloses address a. If specified size s exceeds byte sequence $\mu(seg, a_{base})$ beginning from a, ϵ^a has to take direct successor byte sequences within seg into account to be defined. If size s is not a multiple of 8, the resulting byte sequence will be constructed by adding additional high order 0 bits until its bitsize reaches the next higher multiple of 8.

Using ϵ^a, we now construct a function to read raw byte data from memory using selectors. We define $\epsilon^s : S_G \times Selectors : BYTE^*$ as

$$\epsilon^s((\nu, \mu), sel) =_{\text{def}} \epsilon^a((\nu, \mu), \nu^-(sel) + \omega(sel), \sigma(\tau(sel)))$$

We now define a function $\epsilon^e : S_G \times Expr \not\to BYTE^*$, which evaluates a given GIMPLE expression according to the current memory valuation. As GIMPLE expressions contain at most one operator, we can do this by distinguishing different expression types. For expressions consisting of constant values or selectors, ϵ^e corresponds to applications of ι^- or ϵ^s respectively. Other types of expressions may be evaluated using one of the following definitions of ϵ^e:

Let $\square \in \{+, -, *, /, \%, \wedge, \vee, >, <, \geq, \leq, =, \neq\}$ be a binary arithmetic or boolean operator, and let $exp =_{\text{def}} exp_1 \square exp_2$ be an application to two operand expressions. We define

$$\epsilon^e((\nu, \mu), exp) =_{\text{def}} \\ \iota^-(\iota(\epsilon^e((\nu, \mu), exp_1), \tau^*(exp_1)) \square \iota(\epsilon^e((\nu, \mu), exp_2), \tau^*(exp_2)), \tau^*(exp))$$

Concurrently, for an unary arithmetic or boolean operator $\diamond \in \{+, -, !\}$ and expression $exp =_{\text{def}} \diamond exp_1$ we define

$$\epsilon^e((\nu, \mu), exp) =_{\text{def}} \iota^-(\diamond \iota(\epsilon^e((\nu, \mu), exp_1), \tau^*(exp_1)), \tau^*(exp))$$

For a bitwise operator $\circ \in \{\&, |, \mathbf{XOR}\}$ and expression $exp =_{\text{def}} exp_1 \circ exp_2$, the operation is performed on raw byte data, and we define

$$\epsilon^e((\nu, \mu), exp) =_{\text{def}} \epsilon^e((\nu, \mu), exp_1) \circ \epsilon^e((\nu, \mu), exp_2)$$

For bitwise unary operator $\sim$ and according expression $exp =_{\text{def}} \sim exp_1$, we define

$$\epsilon^e((\nu, \mu), exp) =_{\text{def}} \sim \epsilon^e((\nu, \mu), exp_1)$$

For a shift operator $\triangle \in \{\ll, \gg\}$ and expression $exp =_{\text{def}} exp_1 \triangle exp_2$, the opera-

tion is performed as follows (note that exp_2 must correspond to an integral type):

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \epsilon^e((\nu,\mu), exp_1) \triangle \iota(\epsilon^e((\nu,\mu), exp_2), \tau(exp_2))$$

Dereferencing of a selector, $exp =_{\text{def}} *sel$, may be evaluated using

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \epsilon^a((\nu,\mu), \iota(\epsilon^s((\nu,\mu), sel), \mathbb{N}_0), \sigma(\vec{\tau}(sel)))$$

Conversely, referencing of a selector, $exp =_{\text{def}} \&sel$, is defined as

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \iota^-(\nu^-(sel) + \omega(sel), \mathbb{N}_0)$$

For a cast expression $exp =_{\text{def}} (t)(exp_1)$ with target type t, we define

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \iota^-((t)_C\, \iota(\epsilon^e((\nu,\mu), exp_1), \tau^*(exp_1)), t)$$

where cast operator $()_C$ uses C cast operator semantics for atomic types t and $\tau^*(exp_1)$.

For purposes of legibility, we henceforth denote ϵ^e by ϵ unless noted otherwise. For specifying the effect of write operations on the memory, we use function

$$\phi : (\mathbb{N}_0 \times \mathbb{N}_0) \times S_G \times BYTE^* \nrightarrow (Seg \times \mathbb{N}_0 \nrightarrow BYTE^*)$$

To begin with, function application $\phi((a_{tgt}, o_{tgt}), (\nu,\mu), val_{byte})$ determines the target memory segment for target base address a_{tgt} and offset o_{tgt}. It then returns a new memory valuation μ', which differs from μ only in the new valuation of the target segment, starting at target base address a_{tgt} but unchanged before offset o_{tgt}. Starting at the offset, the memory is changed according to the byte sequence val_{byte}.

3.4 Transition relation: operational rules.

The operational rules specifying the transition relation $\longrightarrow_G \subseteq S_G \times S_G$ on the GIMPLE state space are based on the control flow graph representation of each GIMPLE function. In Plotkin-style notation, each rule is of the form

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2,\ \iota(\epsilon((\nu,\mu), g), int) \neq 0}{(n_1,\nu,\mu) \longrightarrow_G (n_2,\nu',\mu')}$$

Informally speaking, a transition $(n_1,\nu,\mu) \longrightarrow_G (n_2,\nu',\mu')$ is possible if (1) there exists an edge from n_1 to n_2 in the respective CFG, (2) the guard condition g associated with this edge evaluates to `true` (for C-like languages this means that it evaluates to an integral value not equal to zero) in the current valuation (ν,μ). For each type of statement encoded in the nodes n_1 it remains to define the *effect* of this statement on (ν,μ), resulting in the new valuation state (ν',μ'). Below we give some examples of detailed rule specifications.

(1) The effect of a *stack variable definition*, $n_1 =_{\text{def}}$ `typex x;`, is to allocate the required space on stack. The values, however, are still undefined. As a consequence the effect on the memory valuation can be specified by

$$\nu' = \nu \oplus [(stack, a') \mapsto x]$$

where a' is a fresh address not occurring in dom ν (in fact, we use the proper offset of x from the base of the stack frame for building a'). The effect on the memory is

$$\mu' = \mu \oplus [(stack, a') \mapsto \underbrace{\langle ?, \ldots, ? \rangle}_{\texttt{sizeof(typex)}}]$$

where "?" denotes that the byte values are still undefined.

(2) The effect of an *assignment to a selector*, $n_1 =_{\text{def}} \texttt{sel} = \texttt{expr};$, is to change the memory at the base address plus offset, as defined by the selector according to the expression valuation. As the left-hand and right-hand sides of the assignment need not necessarily be typed identically, we first construct the artificial cast expression $expr' = (\tau(sel))(expr)$. As we have now ensured $expr'$ to be of the type corresponding to sel, we go on and assign

$$\nu' = \nu,\ \mu' = \phi((\nu^-(sel), \omega(sel)), (\nu, \mu), \epsilon((\nu, \mu), expr'))$$

(3) The effect of an *assignment to a de-referenced selector*, $n_1 =_{\text{def}} *\texttt{sel} = \texttt{expr};$, is to change the memory at the address pointed to by `sel` according to the expression valuation and the pointer target type of the selector. We therefore need to calculate the target address a_{trg} of the write operation first. This is done by evaluating $a_{trg} = \iota(\epsilon^s((\nu, \mu), sel), \mathbb{N}_0)$. Again using an artificial cast expression $expr' = (\vec{\tau}(sel))(expr)$, we can now construct a new state space by assigning

$$\nu' = \nu,\ \mu' = \phi((a_{trg}, 0), (\nu, \mu), \epsilon((\nu, \mu), expr'))$$

(4) The effect of copying memory, $n_1 =_{\text{def}} \texttt{memcpy(trg, src, s)};$, is to copy `s` successive bytes starting with address `src` to the memory indicated by `trg`. This may be accomplished by defining

$$a_{src} = \iota(\epsilon((\nu, \mu), src), \mathbb{N}_0)$$
$$a_{trg} = \iota(\epsilon((\nu, \mu), trg), \mathbb{N}_0)$$

to be the addresses specified in `src` and `trg` respectively. We can now construct

$$\mu_0 = \phi((a_{trg}, 0), (\nu, \mu), \epsilon^a((\nu, \mu), a_{src}, 8))$$
$$\dots$$
$$\mu_i = \phi((a_{trg}, 8 * i), (\nu, \mu_{i-1}), \epsilon^a((\nu, \mu_{i-1}), a_{src} + 8 * i, 8))$$
$$\dots$$
$$\mu_{s-1} = \phi((a_{trg}, 8 * (s-1)), (\nu, \mu_{s-2}), \epsilon^a((\nu, \mu_{s-2}), a_{src} + 8 * (s-2), 8))$$

and finally

$$\nu' = \nu$$
$$\mu' = \mu_{s-1}$$

(5) The effect of a *function invocation*, $n_1 =_{\text{def}} \texttt{sel} = \texttt{f(x1, \dots, xn)};$, for a function with prototype `t f(t1 z1,...,tk zk)` is calculated according to the following operational rule:

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2,\ \iota(\epsilon((\nu, \mu), g), int) \neq 0,\ (I(G_f), \nu_1, \mu_1) \longrightarrow_G^* (O(G_f), \nu_2, \mu_2)}{(n_1, \nu, \mu) \longrightarrow_G (n_2, \nu', \mu')}$$

In this rule, ν_1, μ_1 are extensions of ν, μ which comprise the initial settings of the formal parameters and the return value:

$$\nu_1 = \nu[(stack, a) \mapsto xReturn, (stack, a_1) \mapsto z_1, \dots, (stack, a_k) \mapsto z_k]$$

Here $a, a_1, \dots, a_k$ are fresh address values and $xReturn$ is an auxiliary name for the stack location storing the return value. The initial valuation of $xReturn$ is

undefined, but the z_i carry the valuation of their actual parameters x_i:

$$\begin{aligned}
&\mu_1 = \mu_1^1\\
&\mu_1^1 = \phi((\nu_1^-(z_1),0),(\nu_1,\mu_1^2),\epsilon((\nu_1,\mu_1^2),(t_1)(x_1)))\\
&\dots\\
&\mu_1^i = \phi((\nu_1^-(z_i),0),(\nu_1,\mu_1^{i+1}),\epsilon((\nu_1,\mu_1^{i+1}),(t_i)(x_i)))\\
&\dots\\
&\mu_1^k = \phi((\nu_1^-(z_k),0),(\nu_1,\mu_1^{k+1}),\epsilon((\nu_1,\mu_1^{k+1}),(t_k)(x_k)))\\
&\mu_1^{k+1} = \mu[(stack,a) \mapsto \underbrace{\langle ?,\dots,? \rangle}_{\texttt{sizeof(t)}}]
\end{aligned}$$

Now the precondition $(I(G_f),\nu_1,\mu_1) \longrightarrow_G^* (O(G_f),\nu_2,\mu_2)$ in the operational rule above requires that a sequence of transitions through the CFG of f should exist, starting with valuation ν_1,μ_1, so that the final valuation before function return, ν_2,μ_2, defines the target state (n_2,ν',μ') via

$$\mu' = \phi((\nu_2^-(sel),\omega(sel)),(\nu_2,\mu_2),\epsilon((\nu_2,\mu_2),(\tau(sel))(xReturn)))$$

Finally, the local variable addresses and associated memory valuations of f are removed from ν',μ'.

4 Symbolic Interpretation of GIMPLE-Programs

For symbolic interpretation the state space is defined as

$$\begin{aligned}
&S_S = N(P) \times \mathbb{N}_0 \times M\\
&M = dataSegment \times heapSegment \times stackSegment\\
&dataSegment = M\text{-}Item^*\\
&heapSegment = M\text{-}Item^*\\
&stackSegment = stackFrame^*\\
&stackFrame = M\text{-}Item^*\\
&M\text{-}Item = \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\}) \times BaseAddress \times\\
&\qquad Types \times Offset \times Length \times Value \times Constraint\\
&BaseAddress = String\\
&Offset = Length = Value = Constraint = Expr(Symbols_S)\\
&Symbols_S = Symbols \times \mathbb{N}_0
\end{aligned}$$

Each symbolic state consists of a triple $(node,n,mem)$ where $node$ is a node in the GIMPLE control flow graph representing the current "program counter state" of the symbolic execution, n serves as an instruction counter and mem is the current history state of symbolic memory valuations, called *memory items* $m \in mem$. The collection of memory items generated so far is structured according 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.

The components of a memory item are accessed using $m.v_0$, $m.v_1$, $m.a$, $m.t$, $m.o$, $m.l$, $m.val$, $m.c$ for the respective projections. Component $m.a$ represents the base address of a memory item, typically denoted by $\&x$ if the memory location corresponds to a variable x or by a fictitious address symbol representing the start address of a dynamic memory allocation. Component $m.o$ denotes the offset from the base address, where the value specified in $m.val$ is written to. For writing one value $m.val$, the memory portion starting at $m.a + m.o$ is used, and the length of

this portion is determined by the type information $m.t$. If the length specification $m.l$ is a multiple of `sizeof(t)` this specifies that $m.l/$`sizeof(t)` copies of $m.val$ are written into the respective memory segment, starting at $m.a+m.o$. Component $m.c$ represents a symbolic validity constraint for the existence of the item. For concrete or abstract interpretations this means that the memory item is only feasible if $m.c$ – after having been properly resolved – evaluates to true.

For the symbolic specification of offsets, lengths, values and constraints GIMPLE expressions over symbols from $Symbols_S$ are used: Such an expression addresses each identifier as a pair (x, n) where $x \in Symbol$ is an ordinary GIMPLE symbol and n is a version information. Components $m.v_0, m.v_1$ represent validity information: When resolving a symbol $(x, n) \in Symbols_S$ occurring in offset, length, value or constraint expressions of some memory item m', only the items m with $m.v_0 \leq n \leq m.v_1$ are considered.

In symbolic interpretation expressions are never resolved to concrete or abstracted variable values, instead, a resolution stops if the expression only contains literals (including base addresses which are considered as string literals), operators and symbols from a given set V and with a specific version range $n_0 \leq n \leq n_1$. A typical resolution variant is to take V as the set of base addresses, function call parameters and global input variables, and specify $n = 0$, meaning *"resolve expression until it only contains literals, operators and input variables in their initial version"*. The constraints of the memory items involved are part of the resolution result ρ, so in general ρ is of the form

$$\rho = \textbf{if } c_{11} \wedge \ldots \wedge c_{1k_1} \textbf{ then } e_1 \textbf{ elseif } c_{21} \wedge \ldots \wedge c_{2k_2} \textbf{ then } e_1 \ \ldots \textbf{else } e_\ell$$

with expressions $e_i \in Expr(V)$, that is, without version information. Examples for handling memory items in S_S are given in Section 6.

Symbolic interpretation 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 [5]. To illustrate the effect of symbolic transitions on the state space S_S we present three transition rules explaining stack variable definition, assignment to a variable (selector) and assignment to a de-referenced pointer.

(1) A *stack variable definition* , $n_1 =_{\text{def}}$ `typex x;` only affects the current stack frame. Value expression $\top$ marks that the value is still undefined.

$$mem' = (mem.data, mem.heap, front(mem.stack) \frown \langle last(mem.stack) \frown \langle m \rangle \rangle$$
$$m = (n+1, \infty, \&x, \texttt{typex}, 0, 8 \cdot \texttt{sizeof(typex)}, \top, (g, n))$$

(2) The effect of an *assignment to a stack variable*, $n_1 =_{\text{def}}$ `sel = expr;` affects the current stack frame only:

$$mem' = (mem.data, mem.heap, front(mem.stack) \frown \langle h' \rangle)$$
$$h' = up_{=}(sel, expr, n, last(mem.stack), g)$$

[5] It may turn out, however, on 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`.

function $up_{=}(sel : Selectors;\ expr : Expr;\ n : \mathbb{N}_0;\ h : M\text{-}Item^*;\ g : Expr) : M\text{-}Item^*$
$m' := (n+1, \infty, \&\beta(sel), \tau(stack, sel), \omega_A(sel), 8 \cdot \texttt{sizeof}(sel), (expr, n), (g, n));$
$up_{=} := up(m', n, h);$
end

Fig. 2. Effect of normal assignments on history of memory items.

Function $up_{=}()$ (Fig. 2) specifies (1) how a new memory item m' is created for the stack frame history, carrying the right-hand side expression as its value and the CFG guard condition as validity constraint and (2) 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 function $up()$ (Fig. 3): All memory items m matching with the new item m' with respect to base address and validity information have to be invalidated. It may be the case, however, that m' "overwrites" only a portion of m. As a consequence, it has to be specified that the "remains" of m not affected by the assignment m' are still valid. Therefore a new memory item m_1 is created and its constraint specifies that outside the range of m', the old m valuation still exists. Observe that the constraint of m_1 always evaluates to `false` if m and m' are of the same type and have the same offset. This indicates that m_1 is infeasible, so m is completely overwritten.

The effect of assignments to variables in the data segments are specified analogously; they affect the *mem.data*-portion of the memory state.

(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. 4).

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

At first, a list ml of all possible pointer targets is generated, using auxiliary function $\gamma()$ (Fig.5): Depending on the valuation of different constraints associated with different memory items, p may point to one or more locations in stack, data segment or heap. For each of these possible situations, ml contains the new memory item for the respective pointer target. The effect of each new item on the invalidation of existing items and creation of new ones is performed again as specified by $up()$ and explained above.

5 Abstract Interpretation of GIMPLE-Programs

Based on the symbolic interpreter introduced in the preceding section it is now possible to construct a *variety of abstract interpreters* according to the following rules:

(1) For every datatype t in the concrete program component chose a suitable abstraction lattice $(L(t), \sqsubseteq)$, so that a Galois connection $(\mathbf{P}(t), \subseteq) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} (L(t), \sqsubseteq)$ exists.

(2) Lift each operation $\diamond$ defined on t to $L(t)$ by means of the canonic con-

function $up(m' : \textit{M-Item};\ n : \mathbb{N}_0;\ h : \textit{M-Item}^*) : \textit{M-Item}^*$
  $u := \langle\,\rangle;\ w := \langle\,\rangle;$
  for $m = last(h)$ **downto** $head(h)$ **do**
    if $(m.v_1 = \infty \wedge m'.a = m.a)$ **then**
      $m_1 := (n+1, \infty, m.a, m.t, \underline{o}, \underline{l}, m.val,$
        $m.c \wedge m'.c \wedge 0 < \underline{l} \wedge m.o \leq \underline{o} \wedge \underline{o} < \underline{l} \wedge \underline{l} \leq m.l \wedge (\underline{o} + \underline{l} \leq m'.o \vee m'.o + m'.l \leq \underline{o}));$
      $m.v_1 := n;$
      $u := \langle m_1 \rangle \frown u;$
    endif
    $w := \langle m \rangle \frown w;$
  enddo
  $up := w \frown u \frown \langle m' \rangle;$
end

Fig. 3. Effect of new memory item on history $h \in M^*$.

function $up_{=_p}(p : \textit{Symbols};\ \textit{expr} : \textit{Expr};\ n : \mathbb{N}_0;\ \textit{mem} : M;\ g : \textit{Expr}) : M$
  $h_d := mem.data;\ h_h := mem.heap;\ h_s := last(mem.stack);$
  $ml := \gamma(p, expr, n, mem, g);$
  forall m' **in** ml **do**
    if $(\sigma(m') = data)$ **then** $h_d := up(m', n, h_d)$
    elseif $(\sigma(m') = heap)$ **then** $h_h := up(m', n, h_h)$
    else $h_s := up(m', n, h_s);$
  enddo
  $up_{p=} := (h_d, h_h, front(mem.stack) \frown \langle h_s \rangle);$
end

Fig. 4. Effect of assignments to de-referenced pointers on history of memory items.

function $\gamma(p : \textit{Symbols};\ \textit{expr} : \textit{Expr};\ n : \mathbb{N}_0;\ \textit{mem} : M;\ g : \textit{Expr}) : \textit{M-Item}^*$
  $ml := \langle\,\rangle;$
  if $\sigma(p) = data$ **then** $h := mem.data$ **else** $h := last(mem.stack);$
  forall m_p **in** h **do**
    if $m_p.a = \&p \wedge m_p.v_0 \leq n \leq m_p.v_1$ **then**
      $pl := \xi(m_p.val, mem);$
      forall q **in** pl **do**
        $a :=$ base address from expression pl;
        $o :=$ offset expression from expression pl;
        $c :=$ conjunction over all conditions of memory items occurring in pl;
        $m' := (n+1, \infty, a, \vec{\tau}(p), o, 8 \cdot \texttt{sizeof}(\vec{\tau}(p)), (expr, n), c \wedge (g, n));$
        $ml := ml \frown \langle m' \rangle;$
      enddo
    endif
  enddo
  $\gamma := ml;$
end

Fig. 5. Function γ finds list of memory items potentially affected by assignment to de-referenced pointer.

```
function ξ((expr, n) : Expr × ℕ0; mem : M) : M-Item − Expr*
  el := ⟨expr⟩;
  e := head(el);
  while e is not resolved to base address plus M-Item-expression for offset do
    x := next unresolved identifier from e;
    h := if σ(x) = stack then last(mem.stack) else mem.data;
    for m := last(h) downto head(h) do
      if m.a = &β(x) ∧ m.v0 ≤ n ≤ m.v1 then
        e1 := e;
        In e1: exchange each occurrence of x by m;
        el := el ⌢ ⟨e1⟩;
      endif
    enddo
    el := tail(el);
    e := head(el);
  enddo
  ξ := el;
end
```

Fig. 6. Function ξ finds list of base addresses potentially associated with a pointer.

struction $\diamond_L : L(t) \times L(t) \to L(t)$; $p_1 \diamond_L p_2 =_{\text{def}} (p_1{}^{\triangleleft} \diamond_{\mathbf{P}} p_2{}^{\triangleleft})^{\triangleright}$. 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) Lift the symbolic state space $S_S = N(P) \times \mathbb{N}_0 \times M$ defined above to its lattice representation $S_L = N(P) \times \mathbb{N}_0 \times L(M)$, where $L(M)$ is the interpretation of memory items over the respective abstraction lattices chosen for offsets, length, values and constraints.

(5) The transition rules for the abstract interpretation semantics over S_L are of the form

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2, (n_1, n, mem) \longrightarrow_G (n_2, n+1, mem'), L(mem) \models_L (g \neq \texttt{false})}{(n_1, n, L(mem)) \longrightarrow_L (n_2, n+1, L(mem'))}$$

where $L(mem)$ denotes the lattice interpretation of memory items. Informally speaking, an abstract transition between CFG nodes n_1 and n_2 with changes in abstract memory valuations from $L(mem)$ to $L(mem')$ is possible in S_L if (a) there exists a corresponding edge in the CFG, (b) the lattice valuation of the guard condition g is `true` or $\top$ and (c) the collection of memory items changes from mem to mem' in the symbolic interpretation.

6 Application Example

The following example illustrates some of the advantages obtained by the higher flexibility resulting from the interplay between symbolic and abstract interpretation.

Consider the GIMPLE function[6] shown in Fig. 7 and an associated invocation $x = f(i_0, z_0)$; Applying the symbolic interpretation rules described in Section 4 for the two possible paths through the function results in the symbolic state of the stack frame as shown in the list of memory items on the right-hand side of Fig. 7, valid at function return in line 11. Consider the following verification goals: *(Goal 1): $f()$ only assigns to valid de-referenced pointers.*, *(Goal 2): $f()$ never returns an undefined value.*

```
0   float f(int i, float z) {
1     float *p, *q;
2     float a[10];
3     p = &a;
4     q = p + 4*i;
5     if ( 0 < z ) {
6       *q = 10 * z;
7     else {
8       *q = 0;
9     }
10    return a[i];
11  }
```

Line No. Resulting M-Item

0. $(1, \infty, \&i, int, 0, 32, (i_0, 0), \texttt{true})$
0. $(1, \infty, \&z, float, 0, 32, (z_0, 0), \texttt{true})$
0. $(1, 6, \&xReturn, float, 0, 32, \bot, \texttt{true})$
1. $(2, 3, \&p, float*, 0, 32, \bot, \texttt{true})$
1. $(2, 4, \&q, float*, 0, 32, \bot, \texttt{true})$
2. $(3, 5, \&a, float, 0, 320, \bot, \texttt{true})$
3. $(4, \infty, \&p, float*, 0, 32, \&a, \texttt{true})$
4. $(5, \infty, \&q, float*, 0, 32, (p + 4 \cdot i, 4), (0 \leq i < 10, 4))$
6. $(6, \infty, \&a, float, (32 \cdot i, 5), 32, (10 \cdot z, 5), (0 < z, 5))$
6. $(6, \infty, \&a, float, \underline{o}, \underline{l}, \bot, (0 < z \wedge 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i \vee 32 \cdot i + 32 \leq \underline{o}), 5))$
8. $(6, \infty, \&a, float, (32 \cdot i, 5), 32, 0, (z \leq 0, 5))$
8. $(6, \infty, \&a, float, \underline{o}, \underline{l}, \bot, (z \leq 0 \wedge 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i \vee 32 \cdot i + 32 \leq \underline{o}), 5))$
10. $(7, \infty, \&xReturn, float, 0, 32, (a[i], 6), \texttt{true})$

Fig. 7. GIMPLE Code sample and associated symbolic interpretation result.

Alternative 1: Interpretation with is-defined and interval lattices.

Chose lattice $L_D = (\{\bot, \Delta, \top\}, \sqsubseteq)$ with $\bot \sqsubseteq \Delta \sqsubseteq \top$ as an appropriate abstraction for checking well-definedness of `float z; float a[10];` (Δ stands for *is-defined*, $\bot$ for *is-undefined*). For checking pointer addresses we abstract integers to intervals over $\mathbb{Z}$: $L_I = (\mathbb{I}(\mathbb{Z}), \subseteq)$. With these lattices, we now perform the corresponding abstract interpretation on the history of memory items in Fig. 7, each time resolving the associated to symbols down to constants, base addresses or input variables i_0, z_0 as explained in Section 4. Additionally we assume that a precondition $i_0 \in [3, 5]$ has been asserted. Then the abstract interpretation results in

0. $(1, \infty, \&i, L_I, 0, 32, [3, 5], \texttt{true})$
0. $(1, \infty, \&z, L_D, 0, 32, \Delta, \texttt{true})$ (z is well-defined, since it is initialised with input z_0)
0. $(1, 6, \&xReturn, L_D, 0, 32, \bot, \texttt{true})$
1. $(2, 3, \&p, L_I, 0, 32, [-\infty, +\infty], \texttt{true})$
1. $(2, 4, \&q, L_I, 0, 32, [-\infty, +\infty], \texttt{true})$
2. $(3, 5, \&a, L_D, 0, 320, \bot, \texttt{true})$
3. $(4, \infty, \&p, L_I, 0, 32, [\&a, \&a], \texttt{true})$ (symbolic single-point interval $[\&a, \&a]$)
4. $(5, \infty, \&q, L_I, 0, 32, \&a + 4 \cdot [3, 5], \texttt{true})$ ($([0, 0][\leq][3, 5][<][10, 10])$ is true in L_I
6. $(6, \infty, \&a, L_D, 32 \cdot [3, 5], 32, \Delta, \top)$ ($0 < \Delta$ evaluates to $\top$, $10 \cdot \Delta$ evaluates to Δ over L_D)

[6] Observe that in contrast to C/C++, GIMPLE always uses byte values in pointer arithmetic. As a consequence, we find assignment `q = p + 4*i;` in line 4, whereas we would write `q = p + i;` in the corresponding C/C++ program.

6. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot [3,5] \vee 32 \cdot [3,5] + 32 \leq \underline{o}))$

8. $(6, \infty, \&a, L_D, 32 \cdot [3,5], 32, \Delta, \bot)$ $(\{0\}^{\triangleright} = \Delta)$

8. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot [3,5] \vee 32 \cdot [3,5] + 32 \leq \underline{o}))$

10. $(7, \infty, \&xReturn, L_D, 0, 32, (a[[3,5]], 6), \texttt{true})$ (not yet resolved – see next paragraph)

Now we apply the resolution rules to 10: First it is noted that $a([[3,5]], 6)$ matches all memory items of the form

$$m = (v_0, v_1, \&a, L_D, 32 \cdot [3,5], 32, val, c),\ v_0 \leq 6 \wedge 6 \leq v_1$$

As a consequence the valuation candidates are those from lines 6. and 8. above. We only have to investigate the feasibility of memory items with undefined valuation $\bot$, so it remains to show that

$$0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot [3,5] \vee 32 \cdot [3,5] + 32 \leq \underline{o}) \wedge \underline{o} = 32 \cdot [3,5] \wedge \underline{l} = 32$$

has no solution; this proof obligation is simplified to showing that no solution of

$$[3,5] + 1 \leq [3,5] \vee [3,5] + 1 \leq [3,5]$$

exists. Unfortunately this predicate evaluates to $\top$ in L_I because we can select (different) numbers from $[3,5]$ in each of its occurrences so that the predicate evaluates either to `true` or to `false`. As a consequence it is necessary to perform 2 partitioning steps of the i_0 interval valuation $[3,5]$ into $[3,3]$ $[4,4]$, $[5,5]$, in order to prove that this predicate is always `false`.

Alternative 2: Interpretation with is-defined and predicate lattice.

As we have seen in the discussion of alternative 1 above, the interval lattice is suitable for proving well-definedness of pointer de-referencings but is quite inefficient to prove the crucial step for well-definedness of the return value. We can fix this by taking the solution of verification goal 1 as constructed above, but using another lattice to represent pointer and integer expressions for discharging goal 2: Let L_P the lattice of predicates over programming variables, together with their comparison operators [7]. Use L_D as above. Abstract interpretation now results in

0. $(1, \infty, \&i, L_P, 0, 32, i = i_0, \texttt{true})$

0. $(1, \infty, \&z, L_P, 0, 32, \Delta, \texttt{true})$

0. $(1, 6, \&xReturn, L_D, 0, 32, \bot, \texttt{true})$

1. $(2, 3, \&p, L_P, 0, 32, \bot, \texttt{true})$

1. $(2, 4, \&q, L_P, 0, 32, \bot, \texttt{true})$

2. $(3, 5, \&a, L_D, 0, 320, \bot, \texttt{true})$

3. $(4, \infty, \&p, L_P, 0, 32, p = \&a, \texttt{true})$

4. $(5, \infty, \&q, L_P, 0, 32, q = \&a + 4 \cdot i_0, \texttt{true})$

6. $(6, \infty, \&a, L_D, 32 \cdot i_0, 32, \Delta, \top)$

6. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i_0 \vee 32 \cdot i_0 + 32 \leq \underline{o}))$

8. $(6, \infty, \&a, L_D, 32 \cdot i_0, 32, \Delta, \top)$

8. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i_0 \vee 32 \cdot i_0 + 32 \leq \underline{o}))$

10. $(7, \infty, \&xReturn, L_D, 0, 32, (a[i_0], 6), \texttt{true})$ (not yet resolved – see next paragraph)

[7] More formally, the *quantifier-free Presburger formulae* over program variables are suitable for our purpose because efficient solvers exist for problems of this type [13].

Now, for the resolution of $(a[i_0], 6)$, all memory items of the form

$$m = (v_0, v_1, \&a, L_D, 32 \cdot i_0, 32, val, c),\ v_0 \leq 6 \wedge 6 \leq v_1$$

match, and the condition for returning an undefined value is

$$0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i_0 \vee 32 \cdot i_0 + 32 \leq \underline{o}) \wedge \underline{o} = 32 \cdot i_0 \wedge \underline{l} = 32$$

which – applying the rules on term replacement and arithmetics in L_P – boils down to $i_0 + 1 \leq i_0$ which is obviously `false`.

7 Conclusion

We have described techniques for concrete and abstract interpretation of C/C++ programs represented in GIMPLE, which basically produces a control flow graph model for each C/C++ function or method. The results are implemented in a tool and they are currently applied for integrated module testing and static analysis of safety-critical embedded systems software in the railway and avionic domains. Applications in the field of automotive control are currently prepared; they focus, however, on model-based test case generation. Due to the intermediate model representation of the tool which uses the same class of hierarchic transition systems for code (control flow graph) and model (e. g. UML 2.0 Statechart) representation, the test case generation mechanisms are the same for code-based and model-based testing. Currently a correctness proof for the abstract interpretation semantics constructed according to the rules given in Section 5 is elaborated: We show that application of these rules always result in a valid abstract interpretation semantics according to Definition 2.1.

References

[1] 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.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, 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).

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[4] 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.

[5] 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.

[6] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex bo olean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 2007.

[7] GCC, the GNU Compiler Collection. The GIMPLE family of intermediate representations. See http://gcc.gnu.org/wiki/GIMPLE.

[8] Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen, May 2007.

[9] 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.

[10] 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.

[11] S. Ranise and C. Tinelli. Satisfiability modulo theories. *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, 21(6):71–81, 2006.

[12] 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.

[13] 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.

[14] 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.

[15] Verified Systems International GmbH, Bremen. *RT-Tester 6.2 – User Manual*, 2007.