On Undecidability Results of Real Programming Languages

Raimund Kirner[1], Wolf Zimmermann[2], and Dirk Richter[2]

[1] Vienna University of Technology, Real Time Systems Group, A-1040 Wien, Treitlstr. 3/182-1,`raimund@vmars.tuwien.ac.at`
[2] Martin-Luther Universität Halle-Wittenberg, Institut für Informatik, 06099 Halle/Saale, Germany, `zimmer@informatik.uni-halle.de`

Abstract. Often, it is argued that some problems in data-flow analysis such as e.g. worst case execution time analysis are undecidable (because the halting problem is) and therefore only a conservative approximation of the desired information is possible. In this paper, we show that the semantics for some important real programming languages – in particular those used for programming embedded devices – can be modeled as finite state systems or pushdown machines. This implies that the halting problem becomes decidable and therefore invalidates popular arguments for using conservative analysis.

1 Introduction

Many program analysis approaches are conservative, i.e. they only deliver approximate results. They guarantee positive results but may overestimate the negative side. E.g., worst-case execution time (WCET) analysis has to deliver an upper bound of the execution time, but this bound may be an overestimation of the real WCET [1, 2]. An other example is live-variable analysis (variables at a certain program point that contain values which are needed for further computations). The set of live variables may be overestimated. However for any variable not in this set, it is guaranteed that its value will never be read. The popular argument is that to derive these informations is not computable in the sense of a Turing-Machine. However, undecidability results or computational intractability results such e.g. the halting problem are based on the assumption that every variable contains an integer (or may store values of an infinite range). A closer look to language definitions such as, e.g., *C*, *C++*, *Java*, *Fortran*, *Ada* or *C#* shows that the base types have finite range. In languages such as *C* or *C++* even pointers or references are of finite range. Most compilers map pointers and references to addresses of the target processor, i.e. they are also finite range. Moreover in programming language (restrictions) for embedded devices there are either no pointers or it is explicitly stated that anonymous objects are allocated statically. This also implies that variables containing pointers can only store values of finite range.

Our aim is to show that restricting variables to finite data types leads in many cases (e.g. *C* or *Java*, but not *Ada*) to the decidability of the halting problem and allows exact computation of programming analysis information such as those mentionend above. It is not claimed that it is practically feasible to compute exact program analysis informations, but the argument not to do it cannot be based on undecidability results.

Our approach is as follows, we show for a representative subset of ISO *C* that its semantics as a state transition system is a pushdown automaton. Since reachability, LTL or CTL model checking on pushdown automata is decidable, the halting problem of this subset becomes decidable. Any analysis that can be expressed as a formula in LTL or CTL becomes decidable and can therefore be computed exactly. In general the semantics of the whole *C*-language can be formalized in this way but this would go beyond the limitations of such an article.

In programming languages there are only two reasons for being infinite state: either some variables may store values of an infinite type (as e.g. references for an unlimited memory) or there is no limitation on the recursion depth. Note that an unlimited heap allocation would require infinite reference types. If each variable can only store a finite range of values, any limitation on recursion depth would lead to a *finite* number of states and therefore to a finite state system. Since programming languages for embedded systems often restrict recursion depth and allows only finite types, the semantics of programs in these languages are finite state machines. Interestingly, for some programming languages the halting problem even becomes undecidable if all variables only can store Boolean values (it is possible to simulate a Turing Machine on the run time stack, when reference parameters, recursion, procedure parameters, local procedures and global/local variables are allowed).

2 Pushdown Systems

A *pushdown system* is a tuple $\Pi \triangleq (\Sigma, \Gamma, I, \rightarrow)$ where Σ is a finite set of *states*, Γ is a finite set (the *stack alphabet*), $I \subseteq \Sigma \times \Gamma^*$ is the set of *initial configurations*, and $\rightarrow \subseteq \Sigma \times \Gamma \times \Sigma \times \Gamma^*$ is a finite relation (the *transition rules*). As usual X^* denotes all finite sequences of elements of a set X, ε the empty sequence, xy denotes the concatenation of two sequences $x, y \in X^*$, and $|x|$ the length of the sequence x. An element $(\sigma, \gamma) \in \Sigma \times \Gamma^*$ is called a *configuration* of Π. Let $\kappa \triangleq (\sigma, \gamma\bar{\gamma})$ be a configuration where $\gamma \in \Gamma$. Then the *head of the configuration* κ is the pair $hd(\kappa) \triangleq (\sigma, \gamma) \in \Sigma \times \Gamma$. A *direct derivation* $\Rightarrow_\Pi \subseteq \Sigma \times \Gamma^* \times \Sigma \times \Gamma^*$ is defined by the following inference rule: $\dfrac{(\sigma, \gamma) \rightarrow (\sigma, \gamma')}{(\sigma, \gamma\bar{\gamma}) \Rightarrow (\sigma', \gamma'\bar{\gamma})}$ for all $\bar{\gamma} \in \Gamma^*$.

The *derivation relation defined by* Π is the reflexive transitive closure $\overset{*}{\Rightarrow}_\Pi$ of $\Rightarrow_\Pi$. A configuration (σ, γ) is *final in* Π iff there is no configuration (σ', γ') such that $(\sigma, \gamma) \Rightarrow_\Pi (\sigma', \gamma')$. A configuration κ is *reachable* in Π if there is an initial configuration $\kappa_0 \in I$ such that $\kappa_0 \overset{*}{\Rightarrow}_\Pi \kappa$. The set of reachable configurations of Π is denoted by $Post^*(\Pi)$.

A *run of* Π is a finite or infinite sequence $\kappa \triangleq \kappa_0\kappa_1 \cdots$ of configurations such that $\kappa_0 \in I$ and $\kappa_i \Rightarrow_\Pi \kappa_{i+1}$ for all $i \in \mathbb{N}, i + 1 < |\kappa|$. If κ is finite, the last configuration must be final. It is possible to transform finite runs to infinite runs by adding the transition rules $hd(\kappa) \rightarrow hd(\kappa)$ for all final configurations κ.

3 Deciding the halting problem on *C*

We now show how a reasonable subset of *C* can be mapped to symbolic pushdown systems (thereby defining the semantics of this subset). In contrast to *C*, we do not limit recursion depth in this dialect. Such a limitation would lead to a finite state system, which is a special case of pushdown systems (i.e. the results implied by the assumptions of this section are more general).

Remark 1. In *C* recursion depth is implicitly limited because it is possible to obtain addresses from local variables using the `&`-operator and the language definition of ISO *C* [3] states that addresses are of finite range (depending on the target). Thus, only finitely many local variables can be stored which implicitly limits the depth of the runtime stack.

```
⟨prog⟩     ::= ⟨decl⟩+
⟨decl⟩     ::= ⟨vardecl⟩ | ⟨procdecl⟩
⟨vardecl⟩  ::= ⟨type⟩ identifier [=⟨expr⟩] ;
⟨procdecl⟩ ::= type identifier ([⟨pars⟩]) ⟨block⟩
⟨type⟩     ::= int | void | ⟨type⟩ *
⟨pars⟩     ::= (⟨par⟩ ,)*⟨par⟩
⟨par⟩      ::= ⟨type⟩ identifier
⟨stat⟩     ::= ⟨assign⟩ | ⟨call⟩ | ⟨vardecl⟩ | ⟨if⟩ | ⟨while⟩ | ⟨block⟩ | ⟨return⟩
⟨assign⟩   ::= ⟨des⟩=⟨expr⟩;
⟨call⟩     ::= ⟨name⟩([args]) ;
⟨args⟩     ::= ⟨expr⟩(,⟨expr⟩)*
⟨if⟩       ::= if (⟨expr⟩)⟨stat⟩[else⟨stat⟩]
⟨while⟩    ::= while (⟨expr⟩)⟨stat⟩
⟨block⟩    ::= {⟨stat⟩*}
⟨return⟩   ::= return [⟨expr⟩] ;
⟨expr⟩     ::= ⟨conj⟩(||⟨conj⟩)*⟩
⟨conj⟩     ::= ⟨rel⟩(&&⟨rel⟩)*
⟨rel⟩      ::= ⟨sum⟩[(== | < | <= | != | > | >=)⟨sum⟩]
⟨sum⟩      ::= ⟨term⟩((+ | -)⟨term⟩)*
⟨term⟩     ::= ⟨unexpr⟩((* | / | %)⟨unexpr⟩)*
⟨unexpr⟩   ::= [!|-]⟨factor⟩
⟨factor⟩   ::= ⟨des⟩ | ⟨call⟩ | const
⟨des⟩      ::= **⟨name⟩
⟨name⟩     ::= identifier
```

Fig. 1. Syntax of a *C-K*

Fig. 1 shows the syntax of *C-K*, a representative subset of *C*. The constructs have the traditional semantics as in *C*. One of the declarations must be a function `main`. We assume that the rules of static semantics are satisfied. Table 1 shows some notions on static informations about the program. The details of these notions are discussed in the following.

GLOB	set of global variable identifiers in a program
PROC	set of global procedure identifiers in a program
LOC_p	set of local variables of procedure $\mathtt{p} \in \mathsf{PROC}$
LABEL_p	set of labels associated with the instructions of procedure $\mathtt{p} \in \mathsf{PROC}$
EXPR_p	set of labels associated with expressions in procedure p.

Table 1. Static Information on Programs

We don't consider arrays because their semantics in *C* is defined by pointers and access to its element is defined by pointer arithmetic. We further do not consider structs or unions because accesses to fields can be directly implemented using pointers. A certain amount of the heap can be reserved using `malloc(int)`. It returns an address on the heap. The heap is finite since the number of addresses is finite. We don't allow the address operator. It is therefore not possible to access local variables of stack frames except for the top stack frame. For simplicity, we don't allow assignment expressions (and use assignment statements instead) and assume a left-to-right evaluation order. *Instructions* are statements (except blocks) and expressions. *Declarations* declare entities or procedures. A declaration is *global* if it is declared in a program, otherwise it is called *local*, i.e., it is contained in a block. Such declarations are uniquely associated with a procedure and are variables. We therefore call a variable declaration *local* to a procedure `p` iff it is declared in the block of `p` (sub-blocks are also allowed). For simplicity, we assume that integer variables and pointer variables store (signed) integers and addresses that can be represented by k Bits. BIT^k denotes the set of bit sequences of length k.

Example 1. Fig. 2 shows a *C*-*K*-program recursively computing faculties. It holds $\mathsf{GLOB} = \{\mathtt{n}\}$, $\mathsf{PROC} = \{\mathtt{fak}, \mathtt{main}\}$, $\mathsf{LOC}_{\mathtt{fak}} = \{\mathtt{n}\}$, and $\mathsf{LOC}_{\mathtt{main}} = \{\mathtt{x}\}$ For each instruction, the label is added as superscript. With these labels, it is $\mathsf{EXPR}_{\mathtt{fak}} = \{6, 7, 8, 10, 12, 13, 14, 15, 16, 17\}$ and $\mathsf{EXPR}_{\mathtt{main}} = \{0, 2, 3\}$.

```
int n;
int fak(int n) {
  if$^{9}$ (n$^{6}$<=$^{8}$1$^{7}$) return$^{11}$ 1$^{10}$;
  return$^{18}$ fak$^{15}$(n$^{12}$-$^{14}$1$^{13}$)*$^{17}$n$^{16}$;
}
void main() {
  n=$^{1}$2$^{0}$;
  int x=$^{4}$fak$^{3}$(n$^{2}$);
  return$^{5}$;
}
```

Fig. 2. A *C*-*K*-program

We now show that the semantics of the *C*-subset can be formally defined by a pushdown system $\Pi \triangleq (\Sigma, \Gamma, I, \rightarrow)$. Intuitively, the set Σ of states represents the global variables and the memory and the stack of the pushdown system represents the procedure stack. The stack alphabet Γ defines the set of possible procedure frames. Table 2 shows the notions used for the formal definition of the pushdown system defining the semantics of a *C*-program. The details are explained in the following paragraphs.

STORE_V^k	set of stores for variables V holding sequences of k Bits
MEM_m^n	set of m-Bit addressed memories holding sequences of n Bits
REG_R^k	set of all register assignments for registers R holding sequences of n Bits
FRAME_p	set of stack frames for procedure p.

Table 2. Notions used for the formal definition of the semantics of *C-K*

Stores are used to model the storage for global and local variables. Memories are used to model heaps. Registers are used to store intermediate values when evaluating expressions. This is needed since functions calls are expressions that may have side-effects. Because of recursion, a runtime stack is needed to maintain the currently active procedure calls. The stack frames are the elements of this runtime stack.

Formally, a *store* for a set V of variables (represented by their names) for storing sequences of k bits is a mapping $\sigma : \mathsf{VAR} \to \mathsf{BIT}^k$. A store for variables $x_1, \ldots, x_n$ with values $v_1, \ldots, v_n$ is denoted by $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. The memory is a mapping $mem : \mathsf{BIT}^k \to \mathsf{BIT}^k$. Thus, $\Sigma \triangleq \mathsf{STORE}_{\mathsf{GLOB}}^k \times \mathsf{MEM}_k^k \cup \{err\}$ where $err \notin \mathsf{STORE}_{\mathsf{GLOB}}^k \times \mathsf{MEM}_k^k$ represents a runtime error that causes to terminate exceptionally the program.

Example 2. For the set GLOB in Example 1 $\{\mathtt{n} \mapsto 5\}$ is a store where the value 5 is stored at variable `n`. $\{00 \mapsto 0, 01 \mapsto 1, 10 \mapsto 2, 11 \mapsto 3\}$ is an example of a 2-Bit addressed memory storing at an address a the value a.

Lemma 1. $|\Sigma| = 2^{|\mathsf{GLOB}| \cdot k + 2^k} + 1$

Thus, the number of states is finite.

Intuitively, a stack frame consists of the instruction to be executed next, a store for the local variables, and a store for registers (required for evaluation of expressions). A function result is also stored in a register. For identifying the instructions of a program each instruction is associated with a unique label. In particular, the sets of labels of the procedures are pairwise disjoint. A register assignment of a procedure p is a mapping $\rho : \mathsf{EXPR}_p \to \mathsf{BIT}^k$. A register assignment that assigns values $u_1, \ldots, u_k$ to registers $r_1, \ldots, r_k$ is denoted by $\begin{array}{c|c|c} r_1 & \cdots & r_k \\ \hline u_1 & \cdots & u_k \end{array}$ Thus, a *stack frame for procedure* $\boldsymbol{p}$ is a tuple $(l, \sigma, \rho) \in \mathsf{LABEL}_p \times \mathsf{STORE}_{\mathsf{LOC}_p}^k \times \mathsf{REG}_{\mathsf{EXPR}_p}^k \triangleq \mathsf{FRAME}_p$. Then, the stack alphabet can be defined as

$$\Gamma \triangleq \biguplus_{p \in \mathsf{PROC}} \mathsf{FRAME}_p$$

Example 3. $\left(6, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 4 & 0 & 2 & 9 & -1 & 22 & 47 & 88 & 2 & 9 \end{array}\right)$ denotes a stack frame for procedure `fak` in Fig. 2.

Lemma 2. $|\mathsf{FRAME}_p| = |\mathsf{LABEL}_p| \cdot 2^{k \cdot (|\mathsf{LOC}_p| + |\mathsf{EXPR}_p|)}$ *and* $|\Gamma| = \sum_{p \in \mathsf{PROC}} |\mathsf{FRAME}_p|$

Thus, the stack alphabet is also finite.

first_p	label of the first instruction of a procedure p
next_l	label of instruction being executed after instruction labelled l
yes_l	label of the first instruction in the positive case of a conditional or loop
no_l	label of the first instruction in the negative case of a conditional or loop
$\mathit{odp}_l(i)$	label of i-th operand
$\mathit{par}_p(i)$	i-th parameter of procedure p

Table 3. Control-flow and Data-flow in *C-K*

Each procedure has a uniquely defined first instruction. Assignments, procedure calls, expressions, program have a unique label, and (except for loops, conditionals, and calls, and return statements) have a unique next instruction. The instruction executed after checking a condition of an if- or while-statement depends on the outcome of the decision. Therefore, there are two possibilities for the next instruction. Expressions, if- and while-statements need the values of their operands and conditional expressions, respectively. Table 3 shows the corresponding functions for the control- and data-flow. Note that these functions are statically defined for each program.

Example 4. In Fig. 2, the instruction with label 6 is the first instruction of procedure `fak` and the instruction with label 0 is the first instruction of procedure `main`. Thus, $\mathit{first}_{\mathtt{fak}} = 6$ and $\mathit{first}_{\mathtt{main}} = 0$. The instruction being executed after the instruction at label 0 is the instruction at label 1. Thus, $\mathit{next}_0 = 1$. Fig. 4(a) shows the complete control-flow for the program in Fig. 2. Not that the instruction to be executed after label 9 is the instruction at label 10 if the condition evaluates to a non-zero value and the instruction at label 12 if the condition evaluates to zero. Thus $\mathit{yes}_9 = 10$ and $\mathit{no}_9 = 12$. Fig. 4(b) shows the data-flow (as a use-def-chain) for the program in Fig. 2. E.g. the operands of the instruction at label 14 are at labels 12 and 13. Thus $\mathit{opd}_{14}(1) = 12$ and $\mathit{opd}_{14}(2) = 13$. The operand of the instruction at label 9 is at label 8, i.e. $\mathit{opd}_9(1) = 8$. The operand of the return-instruction at label 11 is at label 10, i.e. $\mathit{opd}_{11}(1) = 10$.

Table 3 shows the transition rules of the program. It uses the notations shown in Table 4. In the initial state, all global variables are initialized with 0, the contents of the memory is uninitialized, the first procedure to be executed is `main`, and the first instruction being executed is the first instruction of `main`. All local variables of `main` are uninitialized, i.e. any store for the local variables of `main` is allowed for the first configuration. The same remark applies for any register

$$
\begin{aligned}
\mathit{Trans}_s &\triangleq \{\mathit{start} \to ((o,m),(\mathit{first}(\texttt{main}),\sigma,\rho)) : m \in \mathsf{MEM}^k_k, \sigma \in \mathsf{STORE}^k_{\mathsf{LOC}_{\texttt{main}}}, \rho \in \mathsf{REG}^k_{\mathsf{EXPR}_{\texttt{main}}}\}\\
\mathit{Trans}_c &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{c}_{L}))\}\\
\mathit{Trans}_v &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{x}_{L})) : \mathit{var}(L) \in \mathit{GLOB}, x = \sigma'(\mathit{var}(L))\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{x}_{L})) : \mathit{var}(L) \in \mathit{LOC}_{\mathit{proc}(L)}, x = \sigma'(\mathit{var}(L))\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{x}_{L})) : x = \mathit{mem}(\rho(\mathit{reg}(\mathit{opd}_L(1))))\}\\
\mathit{Trans}_{\ominus} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{x}_{L})) : x = \ominus(\rho(\mathit{opd}_L(1)))\}, \qquad \ominus \in \{\texttt{!},\texttt{-}\}\\
\mathit{Trans}_{\oplus} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{x}_{L})) : x = \ominus(\rho(\mathit{opd}_L(1)),\rho(\mathit{opd}_L(2))) \neq \mathit{undef}\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to \kappa : \kappa \in \Sigma \times \Gamma^*, \ominus(\rho(\mathit{opd}_L(1)),\rho(\mathit{opd}_L(1))) = \mathit{undef}\},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \oplus \in \{\texttt{+},\texttt{*},\texttt{-},\texttt{/},\texttt{\%},\texttt{==},\texttt{!=},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\}\\
\mathit{Trans}_{\texttt{\&\&}_1} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{no}_L,\sigma',\rho|^{0}_{L})) : \rho(\mathit{opd}_L(1) = 0\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{yes}_L,\sigma',\rho|^{0}_{L})) : \rho(\mathit{opd}_L(1) \neq 0\}\\
\mathit{Trans}_{\texttt{\&\&}_2} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{\rho(\mathit{opd}_L(2))}_{L})) : \mathit{opd}_L(1) \neq 0\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{1}_{L})) : \mathit{opd}_L(1) \neq 0\}\\
\mathit{Trans}_{\texttt{||}_1} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{yes}_L,\sigma',\rho|^{0}_{L})) : \rho(\mathit{opd}_L(1) = 0\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{no}_L,\sigma',\rho|^{0}_{L})) : \rho(\mathit{opd}_L(1) = 0\}\\
\mathit{Trans}_{\texttt{||}_2} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{\rho(\mathit{opd}_L(2))}_{L})) : \mathit{opd}_L(1) = 0\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho|^{1}_{L})) : \mathit{opd}_L(1) \neq 0\}\\
\mathit{Trans}_{\texttt{=}} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma|^{v}_{x},m),(\mathit{next}_L,\sigma',\rho)) : x \triangleq \mathit{opd}_L(1) \in \mathsf{GLOB}, v \triangleq \rho(\mathit{opd}_L(2))\}\\
&\cup((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma'|^{v}_{x},\rho)) : x \triangleq \mathit{opd}_L(1) \in \mathsf{LOC}_{\mathit{proc}(L)}, v \triangleq \rho(\mathit{opd}_L(2))\}\\
&\cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m|^{v}_{\rho(x)}),(\mathit{next}_L,\sigma',\rho)) : x \triangleq \mathit{opd}_L(1) \notin \mathit{GLOB} \cup \mathit{LOC}_{\mathit{proc}(L)}, v \triangleq \rho(\mathit{opd}_L(2))\}\\
\mathit{Trans}_{\texttt{p(x)}} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{first}_p,\sigma'',\rho')(L,\sigma',\rho)) : \rho \in \mathsf{REG}^k_{\mathsf{EXPR}_p}, p = \mathit{opd}_L(1), \sigma'' \in \mathsf{STORE}^k_{\mathsf{LOC}}, \sigma''(\mathit{par}_p(i)) = \rho(\mathit{opd}_L(i+1))\}\\
\mathit{Trans}_{d_1} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma',\rho))\}\\
\mathit{Trans}_{d_2} &\triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma'|^{v}_{x},\rho)) : x = \mathit{opd}_L(1), v = \rho(\mathit{opd}_L(2)\}\\
\mathit{Trans}_i &\triangleq \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{yes}_L,\sigma',\rho)) : \rho(\mathit{opd}_L(1)) \neq 0\} \cup \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{no}_L,\sigma',\rho)) : \rho(\mathit{opd}_L(2)) \neq 0\}\\
\mathit{Trans}_l &\triangleq \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{yes}_L,\sigma',\rho)) : \rho(\mathit{opd}_L(1)) \neq 0\} \cup \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{no}_L,\sigma',\rho)) : \rho(\mathit{opd}_L(2)) \neq 0\}\\
\mathit{Trans}_{r_1} &\triangleq \{((\sigma,m),(L',\sigma'',\rho')(L,\sigma,\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma,\rho))\}\\
\mathit{Trans}_{r_1} &\triangleq \{((\sigma,m),(L',\sigma'',\rho')(L,\sigma,\rho)) \to ((\sigma,m),(\mathit{next}_L,\sigma,\rho|^{v}_{L})) : v = \rho'(\mathit{opd}_{L'}(1))\}\\
\mathit{Trans}_m &\triangleq \{((\sigma,m),(L,\sigma,\rho)) \to ((\sigma,m),\varepsilon) : \mathit{proc}(L) = \texttt{main}\}
\end{aligned}
$$

Fig. 3. Transitions Rules defining the semantics of *C-K*

$o : V \to \mathsf{BIT}^k$	the function where $o(v) = 0$ for all $v \in V$
$\ominus(v)$	result of applying the unary operator $\ominus$ to $v \in \mathsf{BIT}^k$
$\oplus(v_1, v_2)$	result of applying the binary operator $\oplus$ to $v_1, v_2 \in \mathsf{BIT}^k$
$f\|_x^v$	function where $f\|_x^v(y) \triangleq \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$
$var(l)$	variable at label l
$proc(l)$	procedure where $l \in PROC_{proc(l)}$

Table 4. Notations used in Table 3

$next_0 = 1$	$yes_9 = 10$
$next_1 = 2$	$no_9 = 12$
$next_2 = 3$	$next_{10} = 11$
$next_3 = 4$	$next_{12} = 13$
$next_4 = 5$	$next_{13} = 14$
$next_6 = 7$	$next_{14} = 15$
$next_7 = 8$	$next_{15} = 16$
$next_8 = 9$	$next_{16} = 17$
	$next_{17} = 18$

(a) Control-Flow

$opd_1(1) = \mathtt{n}$	$opd_{11}(1) = 10$
$odp_1(2) = 0$	$opd_{14}(1) = 12$
$opd_3(1) = 2$	$opd_{14}(2) = 13$
$opd_4(1) = \mathtt{x}$	$opd_{15}(1) = 14$
$odp_4(2) = 3$	$opd_{17}(1) = 15$
$opd_8(1) = 6$	$opd_{17}(2) = 16$
$opd_8(2) = 7$	$opd_{18}(1) = 17$
$opd_9(1) = 8$	

(b) Data-Flow

Fig. 4. Control– and Data-Flow for the Program in Fig. 2

value. We therefore introduce an *artificial* initial state *start* and the set of start transitions $Trans_s$. The rules in $Trans_c$ show the rules for accessing constants. It changes the value at the register of the current expression to the value of the constant. Similarly, the rules $Trans_v$ for reading values of variables or from the memory (using indirect addresses) change the value of the register associated with the expression to the corresponding value of the global variable (first case), local variable (second case) or memory address (contained in the register of the operand). The rule $Trans_\ominus$ shows the transition rules for the unary operators $\ominus$. According to the ISO *C* language specification, the behavior is undefined, if the result of a mathematical operation is exceptional (such as division by 0) or an overflow occurs. This means that anything can happen (from continuing program execution at any configuration up to exceptional termination). This undefined behavior is modeled by the second set of transition rules of $Trans_\oplus$. The short-circuit operators are associated with two instructions: The first instructions $\mathtt{\&\&}_1$ and $\mathtt{||}_1$ are used for deciding whether the second operand has to be evaluated. The second instructions are used for finally evaluating the second operand. Fig. 5 shows the control- and data-flow of the short-circuit operators.

The execution of assignments stores the value of the right hand side (second operand) to the variable or address defined by the left hand side of the assignment (and thus changes the store for global variables, local variables, or the memory, respectively). The main idea for the transition rules for procedure calls is to push the current procedure frame onto the stack and allocate a new frame for the called procedure. A variable declaration without an initialization expression has no effect. A variable declaration with an initialization expression has the

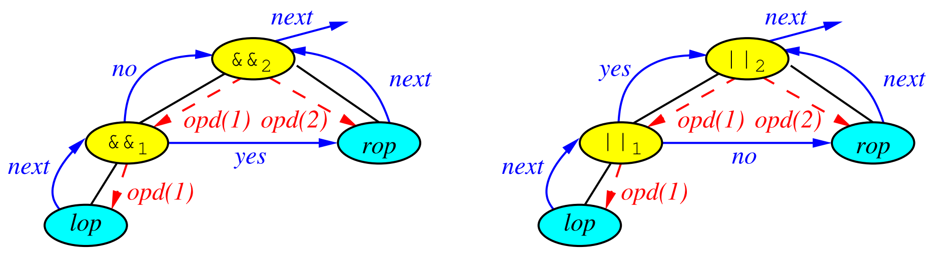

Fig. 5. Control- and Data-Flow for Short-Circuit Operators

same effect as an assignment to this variable (which is always a local variable in this case). The transitions for conditional statements execute the statements of the then-part if the result of the expression is true (beginning at yes_L) and the else-part (if present) or the statement after the conditional (if the else-part is not present), respectively, if the result of the expression is false (in both cases, the execution proceeds at no_L. The transition rule for loops is analogous. The only distinction between conditionals and loops is the definition of yes_L and no_L. The block statement doesn't require a transition rule: If L is the label of the statement preceding the block, then $next_L$ is the first statement within the block. Furthermore, if L is the last statement of the block, $next_L$ is the statement to be executed after the block. The return statement of a proper procedure simply pops the current frame from the stack. The return statement of the function must pass the result of the function. This has been stored at the register for the return expression and must be stored at the register of the function call. Returning from `main` will stop the execution and requires therefore an additional transition rule.

Example 5. Fig. 6 shows a run of the program in Fig. 2. Since we have no pointer variables, we omit the memory.

Remark 2. We have not shown the semantics of allocating objects on the heap m. However, the semantics of `malloc` can easily be implemented by introducing a global variable `void * hp` for storing addresses. It is initialized with the maximal address. With this global variable, it is straightforward to implement `malloc`:

```
void * malloc(int s) {
  hp=hp-s;
  return hp;
}
```

Thus, the semantics of *C-K* is formally described as pushdown system. Since for given configurations κ it is decidable for pushdown systems it is always possible that κ reaches κ' we have the

Corollary 1. *For* C-K*, the halting problem is decidable.*

Remark 3. There is no construct in ISO *C* that prevents modeling of the semantics as a symbolic pushdown system.

start

$$\left(\{n \mapsto 0\}, \left(0, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 20&-1&5\end{array}\right)\right) \qquad \mathit{Trans}_s$$

$$\left(\{n \mapsto 0\}, \left(1, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&-1&5\end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(2, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&-1&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{=}}$$

$$\left(\{n \mapsto 2\}, \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(6, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 4&0&2&9&-1&22&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{fak}(n)}$$

$$\left(\{n \mapsto 2\}, \left(7, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&0&2&9&-1&22&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(8, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&2&9&-1&22&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(9, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&-1&22&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{<=}}$$

$$\left(\{n \mapsto 2\}, \left(12, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&-1&22&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_i$$

$$\left(\{n \mapsto 2\}, \left(13, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&22&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(14, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&47&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{-}}$$

$$\left(\{n \mapsto 2\}, \left(6, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 2&3&7&0&2&1&1&88&2&9\end{array}\right)\left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{fak(n)}}$$

$$\left(\{n \mapsto 2\}, \left(7, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&3&7&0&2&1&1&88&2&9\end{array}\right)\left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(8, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&1&7&0&2&1&1&88&2&9\end{array}\right)\left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(9, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&1&1&0&2&1&1&88&2&9\end{array}\right)\left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{<=}}$$

$$\left(\{n \mapsto 2\}, \left(10, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&1&1&0&2&1&1&88&2&9\end{array}\right)\left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&9&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_i$$

$$\left(\{n \mapsto 2\}, \left(11, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&1&1&0&2&1&1&88&2&9\end{array}\right)\left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&1&2&1&1&88&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(16, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&1&2&1&1&1&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{r_2}$$

$$\left(\{n \mapsto 2\}, \left(17, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&1&2&1&1&1&2&9\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(18, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c}6&7&8&10&12&13&14&15&16&17\\\hline 1&2&0&1&2&1&1&1&2&2\end{array}\right)\left(3, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&5\end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{*}}$$

$$\left(\{n \mapsto 2\}, \left(4, \{x \mapsto -10\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&2\end{array}\right)\right) \qquad \mathit{Trans}_{r_2}$$

$$\left(\{n \mapsto 2\}, \left(5, \{x \mapsto 2\}, \begin{array}{c|c|c}0&2&3\\\hline 2&2&2\end{array}\right)\right) \qquad \mathit{Trans}_{d_2}$$

$$(\{n \mapsto 2\}, \varepsilon) \qquad \mathit{Trans}_m$$

Fig. 6. A run of the program in Fig. 2

3.1 Covering the Platform-specific Semantics of *C*

Modeling the semantics of *C-K* as a pushdown system we made several assumptions about the semantics of some language constructs. This is necessary because the language definition of ISO *C* gives only a partial definition of the functional behavior of *C*. We call this language definition the *platform-independent semantics*. The behavior not defined by the language definition can be freely defined by the platform, where a *platform* is defined as all layers used to execute the program, including the compiler up to the target hardware. We call this completion of the language definition the *platform-specific semantics*. The platform-specific semantics of ISO *C* and its challenges for cross-platform testing have been discussed by Wenzel et al. [4].

The reason why ISO *C* does not define the complete functional behavior of the language constructs is simply because of legacy: while the *C* language was originally developed in 1972 by Kernighan and Ritchie at AT&T Bell Labs using an informal (and incomplete) language definition [5, 6] the ANSI committee to develop the *C* language definition was only formed in 1983. Meanwhile a large pool of *C* compilers with different interpretation of several language constructs had arisen. As there was no simple agreement possible of which existing language interpretation is the "right" one, the ANSI *C* language definition had been restricted to a definition of that partial behavior where most *C* compilers agree. In 1990 the ISO adopted the ANSI *C* language definition as the ISO *C* standard and since then took over the further development of the language definition.

The split of the ISO *C* language definition into a platform-independent semantics and a platform-specific semantics has a serious implication for deciding the halting problem of *C* programs: whether a *C* program halts or not may depend on the platform-specific semantics. Thus, even though the halting problem for a *C* program is decidable for any platform-specific semantics, the halting property can become undefined if no specific platform is assumed.

The following list gives just a few examples where the platform-specific semantics has the potential to influence the halting property of some *C* programs:

- Conversion from floating point to integer types: if the value is outside the range of the integer type the result is undefined.
- Conversion of negative values of an integer type into a smaller signed integer type: if the negative value cannot be represented in the smaller type, the behavior is platform-specific.
- Shift operation: whether the shift operations are signed or unsigned is platform-specific.
- Size of base types: the absolute size of the base types is platform-specific.
- Evaluation order of operands: the evaluation order is platform-specific for most operands. One notable exception are short-circuit operations, where the evaluation order is always from left to right.
- Floating-point operations: the behavior in case of exceptional results like division by zero is undefined.

Everything of the ISO *C* language definition with behavior being undefined or implementation-specific belongs to the platform-specific semantics. An interesting property for ISO *C* is that undefined behavior includes the possibility of non-termination. Thus, we also have the

Corollary 2. *If an ISO* C*-program always terminates according to the language definition, i.e., the platform-independent semantics, then it is free of undefined behavior.*

Deciding the halting problem of a *C* program based only on the platform-independent semantics requires to consider all possible instantiations of the platform-specific semantics. Since for any instantiation of the base type ranges a *C* program has a finite state space and, of course, a finite code length, the set of possible instantiations of the platform-specific semantics is also finite for any ranges of the base types. As a consequence we have the

Corollary 3. *The halting problem of an ISO* C *program based only on the platform-independent semantics and upper bounds of the base type ranges is still decidable (with the result being either* yes, no*, or* platform-specific*).*

Remark 4. The problem complexity is dramatically higher than in case of considering a concrete platform-specific semantics, since each run of a concrete platform-specific semantics is also a run of the platform-independent semantics but not vice versa.

4 Related Work

Software model checking follows a similar approach [7–12]. Mostly, it abstracts behavior to symbolic descriptions, i.e. variables over a finite range etc. can be defined and they form the state space together with control variables. Popular examples or finite state descriptions are PROMELA for the SPIN model-checker [13] and the SMV-model checker [14, 15]. Remopla is a symbolic description for pushdown systems and used for the Moped model-checker [16, 17]. The semantics of Remopla can be formally defined by mapping it to a pushdown system similar as in this work. A special case of pushdown systems are Boolean programs (i.e. each variable can store only Boolean values) [18]. The decidability of reachability of configurations of pushdown-systems is a well-known result that can be found in textbooks [19]. In [20–23] it is shown that also model-checking on pushdown-systems is possible.

The decidability of program analysis at source-code level has already been the subject of investigation. Previously, it has been shown that *may analysis* and *must analysis* are not decidable using ISO C programs as examples [24, 25]. In contrast, we prove the decidability of ISO *C* programs by considering details of the language definition that make ISO *C* programs inherently finite state and thus decidable.

5 Conclusion

The use of conservative program analysis techniques for approximative analysis results is often justified by citing the undecidability of the halting problem.

In this paper we have shown that a class of real programming languages is decidable. Our approach is based on the assumption that the base types of the language and the heap memory are of finite range, but allowing an unbounded recursion depth of function calls. In case of Java the base types are already bounded, but the potentially infinite heap has to be bounded to a maximum size. In case of ISO *C* we have to bound the ranges of base types. The heap in ISO *C* inherently is of finite size for base type ranges because of the address operator and the finite size of pointer variables. Interestingly, Java Byte code has also a heap of finite size because references must be stored in the operand stack and the entries have the lengths of machine words.

We have proven the decidability of such programming languages by showing how to transform them into a pushdown automaton, for which the decidability is a well-known result. As a concrete example we have selected a representative subset of ISO *C*.

Finally, we have got the interesting insight that the halting property for ISO *C* can depend on the concrete system platform, i.e., implementation details not specified by the language definition can influence the halting of a program.

References

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckman, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7** (2008)
2. Kirner, R., Puschner, P.: Obstacles in worst-cases execution time analysis. In: Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing, Orlando, Florida (2008) 333–339
3. Organisation, I.S.: ISO/IEC 9899:1999 Programming Languages - C. 2nd edn. American National Standards Institute, New York (1999) Technical Committe: JTC 1/SC 22/WG 14.
4. Wenzel, I., Rieder, B., Kirner, R., Puschner, P.: Cross-platform verification framework for embedded systems. In: Proc. 5th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'07), Santorini Island, Greece (2007)
5. Kernighan, B.W., Ritchie, D.M.: The C programming language. Prentice Hall Press, Upper Saddle River, NJ (1989)
6. Ritchie, D.M.: The Development of the C Programming Language. In: History of programming languages—II. ACM Press, New York, NY, USA (1996) 671–698
7. Obdrzalek, J. In: Model Checking Java Using Pushdown Systems. LFCS, University of Edinburgh (2002)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G. In: Software Verification with BLAST. In 10th International Workshop on Model Checking of Software (SPIN), LNCS Volume 2648, 235-239. Springer (2003)

9. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F. In: Model Checking Programs. Automated Software Engineering Volume 10(2), 203-232, Kluwer Academic (2003)
10. Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z. In: Model Checking C Programs Using F-SOFT. Proc. of the International Conference on Computer Design (ICCD) (2005)
11. Suwimonteerabuth, D., Schwoon, S., Esparza, J. In: jMoped: A Java Bytecode Checker Based on Moped. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer (2005)
12. Berger, F. In: A test and verification environment for Java programs. Diplomarbeit Nr. 2470, Universitt Stuttgart (2006)
13. G. J. Holzmann: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
14. McMillan, K.: Symbolic Moldel Checking. Kluwer Academic Publishers (1993)
15. Jin Yang, Andreas Tiemeyer: Lazy symbolic model checking. In: DAC 00: Proceedings of the 37th conference on Design automation, ACM Press New York (2000) 35–38
16. Esparza, J., Schwoon, S. In: A BDD-based model checker for recursive programs. LNCS Volume 2102, 324-336, Springer (2001)
17. Schwoon, S. In: Model-Checking Pushdown Systems. Technische Universitt Mnchen (2002)
18. Ball, T., Rajamani, S.K. In: Bebop: A Symbolic Model Checker for Boolean Programs. 7th International SPIN Workshop, LNCS Volume 1885, 113-130, Springer (2000)
19. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. 2nd edn. Addison Wesley (2001)
20. Bouajjani, A., Esparza, J., Maler, O. In: Reachability Analysis of Pushdown Automata: Application to Model-Checking. Proc. of the 8th International Conference on Concurrency Theory, LNCS 1243 (1997)
21. Walukiewicz, I. In: Model checking CTL Properties of Pushdown Systems. In FSTTCS'00, LNCS 1974 (2000)
22. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S. In: Efficient algorithms for model checking pushdown systems. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855 (2000)
23. Esparza, J., Kucera, A., Schwoon, S. In: Model-Checking LTL with Regular Valuations for Pushdown Systems. Proc. of the 4th International Symposium on Theoretical Aspects of Computer Software, LNCS 2215 (2002)
24. Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) **1** (1992) 323–337
25. Ramalingam, G.: The undecidability of aliasing. ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994) 1467–1471