# Towards Semantics-Based WCET Analysis [*]

Mihail Asăvoae[1], Dorel Lucanu[1], Grigore Roşu[2]
[1]Alexandru Ioan Cuza University, Romania
{mihail.asavoae, dlucanu}@info.uiac.ro
[2]University of Illinois at Urbana-Champaign, USA
grosu@illinois.edu

**Abstract**

This paper proposes the use of formal semantics as a basis for worst-case execution time (WCET) analysis. Specifically, the semantics of a RISC assembly language is formally defined using recent advances in rewrite-based semantics, and then is used to discover and eliminate erroneous execution paths in the context of WCET analysis. This paper makes two novel contributions: (1) it shows that using a formal semantics of the employed language can be practically feasible in WCET analysis (not only theoretically desirable); and (2) it shows that the discovery and elimination of erroneous execution paths can not only improve the WCET estimation, but can also be achieved using off-the-shelf technology for rewrite-based semantics.

## 1 Introduction

Ideally, program analysis tools should be based on rigorous semantics of the employed programming languages. Unfortunately, giving a formal semantics (using conventional approaches) to a real language is a non-trivial matter; moreover, even when a semantics is available, it is often not easy to use it for program analysis. Recent research in rewriting logic semantics and in tool development based on such semantics [18, 5] shows encouraging results with respect to both expressiveness and scalability. Moreover, the application of these techniques in the context of real-world low-level languages such as Verilog [13] gives us hope that the theoretically ideal semantics-based approach to program analysis may be, after all, also practically feasible.

We propose a general methodology for worst-case execution time (WCET) analysis centered around a formal executable semantics of the underlying language. We assert that the formal definition of a language has all the necessary information to be used for WCET program analysis and verification. We use the $\mathbb{K}$ rewrite-based semantic framework [18, 5] to define a formal executable semantics for a RISC assembly language, namely an integer restricted fragment of Simplescalar [2]. With it, we can take C programs and "execute" them semantically as follows: first compile them into executables, then extract assembly programs from them using the Simplescalar disassembler [2], then execute the resulting assembly programs in our $\mathbb{K}$ semantics. The choice of the Simplescalar toolset is inspired from [11]. $\mathbb{K}$ is highly-modular, allowing us to start with a high-level semantics of the language and then plugging in the $\mathbb{K}$ description of various micro-architecture such as caches, pipelines or hardware speculation techniques. This way, we specialize the original high-level semantics to one specific to a particular processor, which is what we eventually analyze. The $\mathbb{K}$ tool suite (`http://fsl.cs.uiuc.edu/K`) provides support for concrete and symbolic execution, for state space exploration of concurrent and/or non-deterministic programs, for LTL model-checking, and for full-fledged verification (see also [17]).

[*]This work has been supported by Project POSDRU/88/1.5/S/47646 and by Contract ANCS POS-CCE, O2.1.2, ID nr 602/12516, ctr.nr 161/15.06.2010 (DAK).

To exemplify our semantics-based approach to WCET analysis, we picked a specific but important problem: detection and elimination of erroneous paths, a subset of infeasible execution paths. Indeed, being able to identify such paths in a program and eliminate them from the calculation of the WCET can significantly tighten the estimated WCET bounds.

WCET analysis determines, for all the possible input data, the longest execution path of a program that runs on a particular architecture. Thus, WCET analysis addresses two issues: longest path search and micro-architecture modeling. The former relies on the path analysis ability to discover and eliminate the executions that cannot be exercised under any input, executions called infeasible. Existing solutions for the path analysis problem include static analyses [9, 19, 22] based on abstract interpretation [4], integer linear programming (ILP) approaches [?, ?, 21] and measurement based methods [7, 15]. The longest path search could exhibit a particular problem, with impact on timing bounds tightness and this current work presents only that.

Most of the aforementioned approaches work on assembly language code, extracted from executable files. The path analysis classifies the execution paths into feasible and infeasible. The programs may also exhibit *error execution paths*, either for certain input value or under special conditions (i.e. linking of recompiled code fragments). The most common errors have numerical causes such as overflow/underflow and division by zero, or are memory-related, in case of misaligned accesses. It is important to discover and to use error-related knowledge about programs to improve the timing predictability [20]. For example, the single-path programming technique [16, 10] advocates for predicated code generation, when division by zero errors are possible. Timing analyzers further utilize this predicated instrumentation to improve on timing bounds estimation. It is also possible that the underlying compiler generates preventive code to test if certain numerical errors are possible. The extra tests in the generated code implicitly cover the erroneous paths during the WCET analysis.

There are cases when undetected error paths lead to overestimation of timing bounds. We consider the simple, straight line assembly program, in Figure 1, where, for simplicity, we assume that each instruction executes in one time unit. All the registers, with the exception of `r0` which has value 0, are initialized to symbolic values to exhibit all program behaviors, in this particular case, four possible executions. The longest executable path appears to be when both branches are not taken, which is actually an erroneous path, due to a division by zero at instruction 8. This comes after the following instructions: at program point 1, the register `r2` gets value 1, at line 3, register `r3` is updated with value `1`, which at line 7 is overwritten with value `0`. The division raises an error, the execution stops before the end of the program and the path should be labeled as incorrect, and therefore removed as being the longest executable path.

In this particular work, we rely on $\mathbb{K}$ versatility to define both the programming language and associated analysis methods for erroneous paths detection in WCET estimation. Definitions of certain instructions explicitly state program error conditions such as integer overflow/underflow, division by zero or misalignment. In this way, the erroneous paths are explicitly exposed by the semantic rules. Using the formal semantics, we do not rely on the compiler to generate preventive code, nor on manual instrumentation for error path detection. We use the concrete executable semantics, augmented with timing information, to derive abstract semantics and then we employ reachability analysis techniques to detect and eliminate erroneous paths in the context of WCET analysis.

**Related Work.** We elaborate next on using model checking for WCET estimation. The first use of model checking technology in WCET estimation is introduced in Metzner's paper [14], where a processor with simple cache is checked via multiple runs, selected in a binary search fashion. More recent works are around the UPPAAL model checker [1], where the control flow graph of the program and various micro-architecture features are represented as

```
1.   addi r2, r0, 1;
2.   beq r1, r0, 5;
3.   add r3, r2, r0;
4.   j 6;
5.   add r4, r1, r2;
6.   bne r4, r3, 9;
7.   sub r3, r3, r2;
8.   div r2, r3;
9.   sub r3, r1, r4;
10.
```

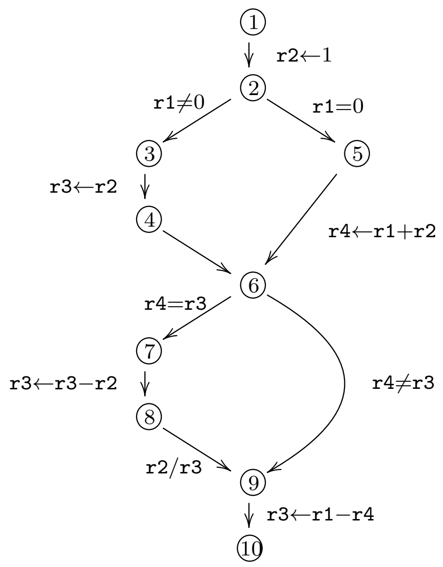

Figure 1: *SSRISC* program (left) with control flow graph (right)

timed automata. In [12], the focus is on multicore systems with timing information extracted from the TDMA and FCFS memory bus models, whereas [6] proposes modular representations of micro-architecture of several ARM processors. In both papers, UPPAAL explores the timed automata models and the WCET of a program is extracted from their clock constraints.

The Maude system [3] is the implementation of rewriting logic and, together with a number of integrated methodologies and tools such as a reachability states exploration tool, an LTL model checker, an inductive theorem prover, as well as other specialized checkers, it enables specification and analysis of programming languages. The $\mathbb{K}$ framework, described in [18], supports the definitions of programming languages using a specialized notation for manipulating program configurations. $\mathbb{K}$ shows its versatily when handling definitions of real languages, such as C in [8] or Verilog in [13] as well as definitions for type systems, model checkers or a Hoare style program verifier [17]. K-Maude tool [5] implements the $\mathbb{K}$ framework on top of Maude system and provides, in this way, access to all Maude's aforementioned supporting tools.

To the best of our knowledge, the methodology we propose is novel in several regards. First, with respect to existing WCET estimation approaches, this is the first use of the formal executable semantics of the underlying language to derive timing bounds. Second, our proposed approach aim at erroneous path discovery to tighten the estimation.

**Outline of the paper.** The paper is organized as follows. Section 2 introduces a specialized framework, called $\mathbb{K}$, for design and analysis of programming languages. We present the K definition for an integer subset of Simplescalar PISA assembly language. Section 3 describes how to use a formal executable semantics to derive abstract semantics for erroneous paths detection in WCET analysis. The last section contains the conclusions.

## 2 The Formal Executable Semantics of *SSRISC* in $\mathbb{K}$

$\mathbb{K}$ [18] is a modular, rewrite-based executable programming language design and semantics framework. A $\mathbb{K}$ specification consists of *configuration*, *computation* and *rule* declarations. A configuration is a (potentially) nested structure formed of $\mathbb{K}$ *cells*. Computation structures, or simply computations, extend the language syntax with a task sequentialization associative operation $\curvearrowright$. The rules in $\mathbb{K}$ are divided into two classes: *computational rules*, that may be

| | | |
|---|---|---|
| `Instr ::=` | `add` *Reg* `,` *Reg* `,` *Reg* `;` | `[strict (2 3)]` |
| | `addi` *Reg* `,` *Reg* `,` *Imm* `;` | `[strict (2)]` |
| | `mult` *Reg* `,` *Reg* `;` | `[strict]` |
| | `div` *Reg* `,` *Reg* `;` | `[strict]` |
| | `j` *Addr* `;` | |
| | `jr` *Reg* `;` | `[strict]` |
| | `beq` *Reg* `,` *Reg* `,` *Addr* `;` | `[strict (1 2)]` |
| | `bne` *Reg* `,` *Reg* `,` *Addr* `;` | `[strict (1 2)]` |
| | `lw` *Reg* `,` *Off* `(` *Reg* `)` `;` | `[strict (3)]` |
| | `sw` *Reg* `,` *Off* `(` *Reg* `)` `;` | `[strict (3)]` |
| | `break ;` | |

Figure 2: The $\mathbb{K}$ annotated SSRISC language syntax: BNF syntax of SSRISC instrutions on the left and their $\mathbb{K}$ strictness attributes on the right. *Reg*, *Addr*, *Imm* and *Off* are of sort *Int32*.

interpreted as transitions in a program execution, and *structural rules*, that modify a term to enable the application of computational rules.

We give practical insights into the $\mathbb{K}$ framework by defining the integer subset of the Simplescalar [2] PISA assembly language (inspired from MIPS IV). We call this language *SSRISC*. Whereas we defined the entire subset of integer-based instructions, for brevity we only describe a representative snippet of it. Apart from the instruction set presented in [2], a number of pseudo-instructions appear in the executables that we analyze; we have also defined those in $\mathbb{K}$, but we also omit them here. The general methodology for language definitions in $\mathbb{K}$ begins with the (abstract) syntax, determines the configuration, and then gives the semantic rules.

The $\mathbb{K}$ annotated syntax of a subset of the *SSRISC* assembly language is given in Figure 2. The left column shows the abstract syntax, in BNF form, while the right column states the corresponding $\mathbb{K}$ strictness attributes that give the evaluation strategies of the declared operators. More precisely, the `strict` attribute tells that the enlisted operands are reduced to base values of ($\mathbb{K}$ builtin) sort *KResult*. The strictness attributes are actually syntactic sugar in $\mathbb{K}$, compactly encoding a set of structural rules that achieve the same result as reduction via evaluation contexts; this encoding is not needed here, the interested reader is referred to [18] for more details. For example, the `add` instruction is `strict` on the second and third operands, which implies that the last two registers, called sources, are reduced to values before the actual addition takes place and the first, destination register, gets this value. When `strict` appears without arguments, like for `mult`, it means strict in all the operands.

The program configuration is a wrapped multiset of cells, written as $\langle \mathit{cont} \rangle_{\mathsf{lbl}}$, where *cont* is the cell contents (possibly itself a multiset of cells) and $\mathsf{lbl}$ is the cell label. The $\mathbb{K}$ cells hold the necessary semantic infrastructure (registers, instruction cache, memory, etc.). Two cells appear in most $\mathbb{K}$ definitions: a cell whose label is $\top$ that encloses all the other cells, and a cell labeled $\mathsf{k}$ that holds the computation (syntax). The $\mathbb{K}$ configuration for the *SSRISC* language is:

$$\mathit{Configuration}_{SSRISC} \equiv \langle\, \langle K \rangle_{\mathsf{k}}\, \langle \mathit{Reg} \rangle_{\mathsf{pc}}\, \langle \mathit{Reg} \rangle_{\mathsf{lo}}\, \langle \mathit{Reg} \rangle_{\mathsf{hi}}\, \langle \mathsf{Map}[\mathit{Reg} \mapsto \mathit{Val}] \rangle_{\mathsf{regs}}\, \langle\, \mathit{Val} \rangle_{\mathsf{break}}\, \rangle_{\top}$$

The $\mathsf{k}$ cell maintains the current computation, i.e., the current program or fragment of program. The computations, i.e. terms of special sort $K$, are nested list structures of computational tasks. Elements of such a list are separated by an associative operator $\curvearrowright$, as in $s_1 \curvearrowright s_2$, and are processed sequentially: $s_2$ is computed after $s_1$; the identity of $\curvearrowright$ is denoted by "$\cdot$". The cell $\mathsf{pc}$ has the program counter and its value indicates the executing instruction. We opt to represent the program counter in a different cell than the other registers, as it improves the readability of the semantics, especially on conditional and unconditional jumps. $\mathsf{lo}$ and $\mathsf{hi}$

are special registers, required by the `mult` and `div` instructions to hold parts of the computed results. The `regs` cell contains all the other registers and is a mapping from register names to stored values. The program requires, as well, a representation of the main memory, that holds both the program and the necessary data. We detach the memory modeling from that of the registers as we plan to keep our specification modular to accommodate specifications of cache memories. The `break` cell is used, in the strict sense, by the instruction `break` and, in the more general sense, to capture program errors such as overflow or division by zero.

Next, we present the concrete formal executable semantics for the *SSRISC* assembly language and, in the next section, we show how to derive useful abstractions out of it. We introduce the $\mathbb{K}$ rules by means of defining the semantic rules of *SSRISC*. $\mathbb{K}$ rules generalize the usual rewriting rules, in the sense that the $\mathbb{K}$ rule manipulates only parts of the rewrite term, in three different ways: read, write and don't care. $\mathbb{K}$ proposes a bidimensional representation of a rule, with the left-hand side placed above a horizontal line and the right-hand side below.

We capture the execution of each *SSRISC* instruction in a number of succesive steps: instruction request for instruction cache or memory, data request from data cache or memory, actual processing and finally, machine state update. Our design target is to capture the language semantics in a correct and concise manner, and, as a result, we propose *a single rewrite rule per each SSRISC instruction.* To achive this, we extract some common functionality, as general register lookup and update, or use some wrappers as in the case of `pc` register update. Before we describe the instruction semantics, we cover these general rules.

The register lookup and update operations require only the cells `k` and `regs`. If the current computational task is a register lookup, for a register $R$, as shown in the first rule in Figure 3 (left-hand side), the resulting configuration has the corresponding value $I$ of $R$ from the register cell. This $\mathbb{K}$ rule brings a new element of the $\mathbb{K}$ notation, the "don't care" part of a list or multiset term, represented with ellipses.

The `pc` register update consists of the following three cases as shown in the right-hand side, in Figure 3. The first rule represents the automated incrementation before an instruction is fetched. The second rule addresses the case of a mandatory jump and updates the `pc` with a specified target address, *NewV*. The last rule represents the fall-through case of a branch instruction and leaves the value of the `pc` register only with the previous normal incrementation.

All the $\mathbb{K}$ rules for the arithmetic-logic (ALU) instructions, in Figure 4, transform the task in the `k` cell into a register update, using either `updateReg` or `updateLo/Hi`. The former takes two arguments, the register and the value to be written, whereas `updateLo` and `updateHi` require only the value. The `add` and `div` instructions have extra checks as they could lead to errors, from overflow and respectively division by zero. Therefore, the first $\mathbb{K}$ rule states that the `add` instruction with the source registers having values $V_1$ and $V_2$ reduces to an overflow check for the signed addition between these two values and, if necessary, followed by the destination register *Rd* update with the result. The `div` instruction in the `k` cell reduces to a division by zero check for the denominator value and, if necessary, followed by the updates for the `lo` and `hi` registers.

The branch and jump instructions, in Figure 5, transform the task in the `k` cell into a correct `pc` register update, which has the next instruction program counter, as a result of instruction fetch. All these $\mathbb{K}$ rules use the `setPC` operation; with the first argument 1 it overwrites the value in the `pc`, and with 0 leaves it unchanged. The two rules for jump instructions change the program counter register with the values *Addr*, respectively the content of the *Rs* register. For the branch instructions, the "fall-through" and "taken" cases correspond to value 0, respectively 1 as the first argument of `setPC`.

Figure 6 shows the $\mathbb{K}$ rules for load and store. The `load` instruction is reduced to a memory read request via `getd`, which takes as arguments the memory address and the destination register *Rd*. Similarly, the `store` instruction is reduced to a memory write request via `putd`, RULE: $\langle \frac{R}{I} \cdots\rangle_{\mathsf{k}} \; \langle \cdots R \mapsto I \cdots\rangle_{\mathsf{regs}}$

RULE: $\langle \frac{\texttt{updateReg}(I, Rd)}{\cdot} \cdots\rangle_{\mathsf{k}} \; \langle \cdots Rd \mapsto \frac{\_}{I} \cdots\rangle_{\mathsf{regs}}$

RULE: $\langle \frac{\texttt{incPC}(PC)}{\cdot} \cdots\rangle_{\mathsf{k}} \; \langle \frac{PC}{PC +_{\texttt{Int32}} 4} \rangle_{\mathsf{regs}}$

RULE: $\langle \frac{\texttt{setPC}(1, NewV)}{\cdot} \cdots\rangle_{\mathsf{k}} \; \langle \frac{\_}{NewV} \rangle_{\mathsf{pc}}$

RULE: $\langle \frac{\texttt{setPC}(0, NewV)}{\cdot} \cdots\rangle_{\mathsf{k}}$

Figure 3: Rules for register look-up and update (left) and pc update (right)

RULE: $\langle \frac{\texttt{add}\ Rd\ ,\ V_1\ ,\ V_2\ ;}{\texttt{ovf}(V_1, V_2) \curvearrowright \texttt{updateReg}(V_1 +_{\texttt{Int32}} V_2, Rd)} \cdots\rangle_{\mathsf{k}}$

RULE: $\langle \frac{\texttt{addi}\ Rd\ ,\ V_1\ ,\ V_2\ ;}{\texttt{updateReg}(V_1 +_{\texttt{Int32}} V_2, Rd)} \cdots\rangle_{\mathsf{k}}$

RULE: $\langle \frac{\texttt{mult}\ V_1\ ,\ V_2\ ;}{\texttt{updateLo}(V_1 *_{\texttt{Int32}} V_2 \,\%_{\texttt{Int32}} 1 \ll_{\texttt{Int32}} 32) \curvearrowright \texttt{updateHi}(V_1 *_{\texttt{Int32}} V_2 \,/_{\texttt{Int32}} 1 \ll_{\texttt{Int32}} 32)} \cdots\rangle_{\mathsf{k}}$

RULE: $\langle \frac{\texttt{div}\ V_1\ ,\ V_2\ ;}{\texttt{div0}(V_2) \curvearrowright \texttt{updateLo}(V_1 /_{\texttt{Int32}} V_2) \curvearrowright \texttt{updateHi}(V_1 \%_{\texttt{Int32}} V_2)} \cdots\rangle_{\mathsf{k}}$

Figure 4: Semantics rules for *SSRISC* ALU instructions

which has the memory address and the source register *Rd* as arguments. The caches and the main memory, which are omitted from this presentation, process the `getd` and `putd` requests. The last discussed *SSRISC* semantic rule treats the special `break` instruction. The k cell gets the `last` term that ends the computation, while the special `break` cell updates to reflect a program error. We mention that `last` is also used for normal termination of computation.

## 3 Abstract Semantics for WCET Analysis

The $\mathbb{K}$-semantic rules of some *SSRISC* instructions, i.e. `add` and `div` embed tests to prevent numerical errors such as overflow and division by zero, respectively. The language definition also poses memory-related error checks such as misaligned data accesses, for a double word load instruction - an instruction that is not featured in this work. Next, we describe how abstract semantics for error paths detection can be obtained from the language definition. The first step is to consider the input program variables initialized with an unknown value, that we name *symb*. While the language definition remains unmodified, the abstraction triggers extensions in the support operations. A newly defined module for symbolic 32-bit integer operations replaces the corresponding built-in module used by the concrete semantics. For example, the addition operation between two concrete values, denoted by $+_{Int32}$, is extended to handle *symb* abstract value such that, if any of the operands is symbolic, the result is symbolic. This abstraction indeed enables all the possible executions, as the $\mathbb{K}$ rewrite rules for the branch instructions handle potentially symbolic values.

The program in Figure 1 has four execution paths, all of them could be exercised under appropriate input data. There are two paths having the division instruction, and using this abstraction, we are able to accurately catch only the path going through lines 3 and 7. The second path that could terminate with a division by zero error goes through lines 5 and 7. The value for the denominator register, r3 is unknown as the abstraction does not learn from the conditions of the two branch instructions. The unknown denominator produces two possible behaviors, for zero and non-zero values. The former yields an erroneous path, while the latter turns out to be the longest execution in the program.

$$\text{RULE: } \left\langle \frac{\texttt{j}\ \mathit{Addr}\ ;}{\texttt{setPC}(1, \mathit{Addr})} \cdots \right\rangle_{\mathsf{k}}$$

$$\text{RULE: } \left\langle \frac{\texttt{jr}\ \mathit{Rs}\ ;}{\texttt{setPC}(1, \mathit{Rs})} \cdots \right\rangle_{\mathsf{k}}$$

$$\text{RULE: } \left\langle \frac{\texttt{beq}\ V_1\ ,\ V_2\ ,\ \mathit{Addr}\ ;}{\texttt{setPC}(\texttt{Bool2Int}(V_1 =_{\mathit{Bool}} V_2), \mathit{Addr})} \cdots \right\rangle_{\mathsf{k}}$$

$$\text{RULE: } \left\langle \frac{\texttt{bne}\ V_1\ ,\ V_2\ ,\ \mathit{Addr}\ ;}{\texttt{setPC}(\texttt{Bool2Int}(V_1 \neq_{\mathit{Bool}} V_2), \mathit{Addr})} \cdots \right\rangle_{\mathsf{k}}$$

Figure 5: Semantics rules for *SSRISC* branch and jump instructions

$$\text{RULE: } \left\langle \frac{\texttt{lw}\ \mathit{Rd}\ ,\ \mathit{Off}\ (\ V_1\ )\ ;}{\texttt{updateReg}(\texttt{getd}(\mathit{Off} +_{\texttt{Int32}} V_1), \mathit{Rd})} \cdots \right\rangle_{\mathsf{k}}$$

$$\text{RULE: } \left\langle \frac{\texttt{sw}\ \mathit{Rd}\ ,\ \mathit{Off}\ (\ V_1\ )\ ;}{\texttt{putd}(\mathit{Off} +_{\texttt{Int32}} V_1, \mathit{Rd})} \cdots \right\rangle_{\mathsf{k}}$$

$$\text{RULE: } \left\langle \frac{\texttt{break}\ ;}{\texttt{last}} \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{\_}{1} \right\rangle_{\mathsf{break}}$$

Figure 6: Semantics rules for *SSRISC* load, store and break instructions

X $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (*error path*)
$\surd$ $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10$ (*undetected infeasible* 8 *t.u.*)
X $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (*error path*)
$\surd$ $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow 10$ (7 *t.u.*)
$\surd$ $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 9 \rightarrow 10$ (6 *t.u.*)

The longest executable path in the program is an undetected infeasible path, introduced by the abstraction. Our goal now is to improve the initial simple abstraction to detect the erroneous path that goes through lines 5 and 7. We refine the first abstract semantics to update the values of the registers, when the program execution encounters a branch instruction, as we keep the state information unchanged, and modify only two semantic rules, for the `bne` and `beq` instructions, shown in Figure 7. We enhance new information learning, when one of the compared registers contains exact information that is used to update the other. Informally, the new operation, called `updReg` uses its first argument, the value of the program counter *PC*, to get the branch instruction and identify the registers. Using this abstraction, we are able to identify, for the example program, both erroneous paths.

X $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (*error path*)
X $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (*error path*)
$\surd$ $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow 10$ (7 *t.u.*)
$\surd$ $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 9 \rightarrow 10$ (6 *t.u.*)

We work with the current implementation of the $\mathbb{K}$ framework, called $\mathbb{K}$-Maude. It is developed on top of Maude system and, in this way, it has access to all verification tools that Maude offers: a command for reachability analysis, an LTL model checker, an inductive theorem prover. We choose the first method, and perform reachability analysis on our $\mathbb{K}$ specifications, to determine the WCET bounds. $\mathbb{K}$-Maude takes $\mathbb{K}$ specifications and generates rewrite theories in Maude. A rewrite theory has an underlying equational theory, containing equations and membership statements, plus rewrite rules. A rewrite theory defines an abstract transitional system, where the equations represent, via equivalence classes, the states, while the rewrite rules

$$\text{RULE: } \langle \frac{\texttt{beq}\ V_1\ ,\ V_2\ ,\ Addr\ ;}{updReg(PC, V_1, V_2) \curvearrowright \texttt{setPC}(\texttt{Bool2Int}(V_1 =_{Bool} V_2), Addr)} \cdots\rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}$$

$$\text{RULE: } \langle \frac{\texttt{bne}\ V_1\ ,\ V_2\ ,\ Addr\ ;}{updReg(PC, V_1, V_2) \curvearrowright \texttt{setPC}(\texttt{Bool2Int}(V_1 \neq_{Bool} V_2), Addr)} \cdots\rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}$$

Figure 7: Modified semantics rules for *SSRISC* branch instructions

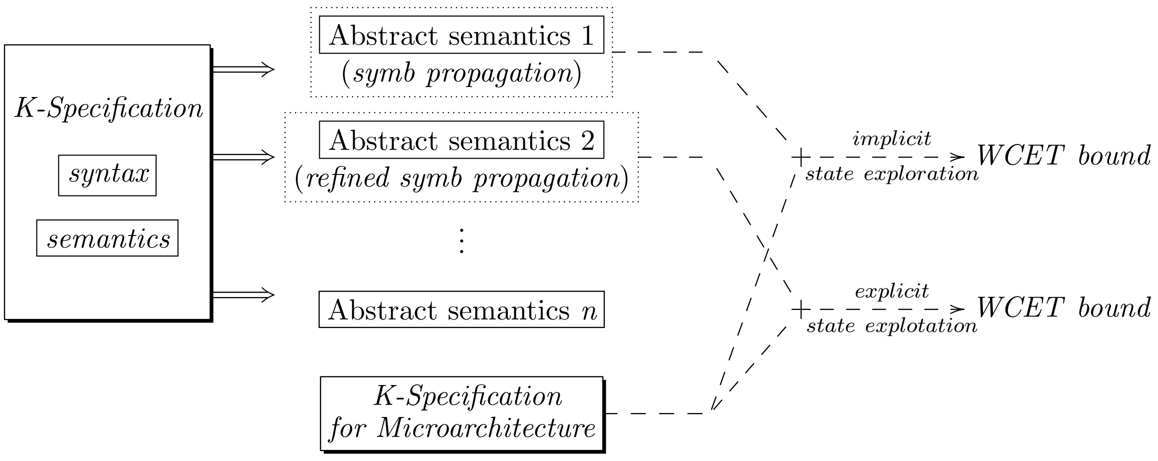

Figure 8: General methodology for WCET estimation, based on the K framework

represent the transitions between these classes. If the left-hand side of a rewrite rule matches the (fragment) of a current state, and the rule condition is satisfied, the system evolves into the state of the right-hand side of the particular rewrite rule. Maude system offers the possibility of unfolding this transition system and proving properties, or getting counterexample information.

To obtain a safe WCET bound for a given program, we need to consider all the possible program executions, implicitly or explicitly. The transition system that a rewrite theory provides could be unfolded using the special `search` command. The class of hard real-time programs assume that program terminates and, in our formal semantics, we denote the final computational task with the token `last`. Therefore, all the program executions should terminate in a state which has `last` in the *k* cell. There are two possible situations: termination when there are no more instructions to be executed and error termination. Both cases are handled via pseudo-instructions that generate a `break` instruction, last rule in Figure 6.

We perform reachability analysis, using the `search` command, for the state that has `last` as the current computational task. The timing information is updated along each execution path and, when the path terminates, the WCET is the maximum of these computed time values. Since we work on straight-line hard real-time programs, the state space exploration guarantees to terminate.

## 4 Conclusions

A WCET analysis determines, for all possible input data, what is the longest program execution on a particular architecture. Thus, the two important issues are: the longest path search, usually based on an annotated control-flow graph of a program and the micro-architecture modeling for a processor behavior analysis. We focused on techniques to improve the longest path search problem. In the context of WCET analysis, our proposed approach is new under two aspects: (1) it is the first framework based on the formal definition of an underlying language in the context of WCET analysis (2) it is the first unitary framework to express both concrete and abstract executions of hard real-time programs. A direct application of (1) is to eliminate erroneous execution paths to improve the results of the WCET estimation.

Our approach towards WCET analysis started with the formal definition of *SSRISC*, an integer subset of Simplescalar PISA assembly language. The concrete executable semantics was defined using $\mathbb{K}$, a rewrite-based definitional framework for design and analysis of programming languages. We used the formal semantics, extended with timing information, to derive two abstract semantics that expose the whole set of program executions, via symbolic values for program variables. During reachability state space exploration, we executed the semantic rules and update the global timing information. We identified and eliminated the erroneous paths, using error prevention mechanisms, presented in the language definition. This approach did not require special program instrumentation, nor rely on the compiler to generate preventive code.

The obvious improvement of our semantics-based WCET analysis is to derive more powerful abstract semantics and to eliminate not only error paths, but much of the infeasible execution paths. We place equal importance on micro-architecture modeling for processor behavior analysis and, in short term, we plan to to integrate $\mathbb{K}$-based pipeline models. These steps converge towards the first WCET analyzer using the rewrite-based technology.

## References

[1] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing uppaal over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.

[2] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25:13–25, June 1997.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[5] Traian Florin Şerbanuţă and Grigore Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In Peter Csaba Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122, 2010.

[6] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. Metamoc: Modular execution time analysis using model checking. In *WCET*, pages 113–123, 2010.

[7] Jean-François Deverge and Isabelle Puaut. Safe measurement-based wcet estimation. In *WCET*, 2005.

[8] Chucky Ellison and Grigore Roşu. A formal semantics of C with applications. Technical Report http://hdl.handle.net/2142/17414, University of Illinois, November 2010.

[9] Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, and David B. Whalley. Bounding loop iterations for timing analysis. In *IEEE Real Time Technology and Applications Symposium*, pages 12–21, 1998.

[10] Raimund Kirner and Peter Puschner. Time-predictable computing. In *Proc. 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, Oct. 2010.

[11] Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.

[12] Mingsong Lv, Guan Nan, Wang Yi, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-Time Systems Symposium*, 2010. forthcoming.

[13] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010.

[14] Alexander Metzner. Why model checking can improve wcet analysis. In *CAV*, pages 334–347, 2004.

[15] Stefan M. Petters. Comparison of trace generation methods for measurement based wcet analysis. In *WCET*, pages 75–78, 2003.

[16] Peter Puschner. The single-path approach towards wcet-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pages 699–704, Dec. 2003.

[17] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST '10)*. LNCS, 2010. forthcoming.

[18] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[19] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[20] Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Design of Systems with Predictable Behaviour*, 2004.

[21] Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In *VMCAI*, pages 309–322, 2004.

[22] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In *CAV*, pages 22–36, 2008.