# On the Modular Integration of Abstract Semantics for WCET Analysis

Mihail Asăvoae(✉) and Irina Măriuca Asăvoae

VERIMAG/UJF, Gières, France
{mihail.asavoae,irina.asavoae}@imag.fr

**Abstract.** We propose here a modular resource analysis which is constructed around a rewrite-based formal specification of an embedded system. Designing and analyzing embedded systems considers both hardware and software behavioral aspects which we capture using the modular notion of system configuration. Hence, we use a configuration-based design methodology and we instantiate parts of the configuration to accommodate data and control-flow abstractions. These instantiations require no modifications of the original formal specification. We implement in this manner a particular resource analysis, namely worst case execution time (WCET), and evaluate it with respect to a reusability metric.

## 1 Introduction

Interaction between an embedded system and the external environment could be stated as a set of constraints, usually produced by resource analyses. For example, a proper scheduling requires an analysis of the time resource, the constraints being generally set in terms of upper/lower bounds. A schedulability analysis requires some of the constraints to be in terms of safe and tight worst case execution time (WCET) bounds. However, to achieve accurate WCET bounds, one has to formalize low-level aspects, e.g., assembly languages and hardware architecture.

The standard workflow for WCET analysis [27] exploits the modularity of the systems (i.e. programs running on specified architectures) by separate analyses for the control-flow and the processor behavior. The control-flow analysis derives flow facts without architectural considerations. The processor behavior analysis computes invariants w.r.t. how instructions behave in the presence of the architecture elements. A successful approach for WCET analysis proposes a mix of integer linear programming (ILP) for path analysis of the control-flow graph (CFG) [19,20] and abstract interpretation (AI) [9] for various architecture elements or their combination [15,18,25,26]. This method (ILP + AI) achieved success through tools as `aiT` [1] which is used in substantial industrial projects, e.g., Airbus certification. Because of the industry's quality requirements, `aiT` is sharpen to produce precise bounds for WCET. However, to the best of our knowledge, the modularity in `aiT` is resumed to the two aspects mentioned above.


U. Dal Lago and R. Peña (Eds.): FOPARA 2013, LNCS 8552, pp. 19–37, 2014.
DOI: 10.1007/978-3-319-12466-7_2

We approach the resource (in particular WCET) analysis workflow from a slightly different perspective, originated from the observation that *a programming language semantics has all the necessary information to define, for any program written in this language, the set of all possible concrete executions.* This view together with the fact that any abstraction is based on (sets of) concrete executions, lead to the idea of a workflow for semantics-based resource analysis. Its core is the *formal executable language semantics* (i.e., basically, a compact representation of the set of concrete executions). The formal aspect of the semantics distills the correct programs from the incorrect ones up to the semantics specification, while the executability aspect allows to thoroughly test such programs and gain confidence in the semantics. These two aspects attest that the language semantics specification forms a *trusted core* which sets a foundation for the integration of abstractions for resource analysis.

We introduce in [3], and use here, the trusted core of a formal executable definition of the MIPS assembly language supported by the Simplescalar toolset [6]. Furthermore, we extend, in [2], the language core with a parametric modeling of instruction caches. In this way, we set the grounds for a standard WCET analysis workflow where the *language semantics* is used in the *control flow analysis* and the *architecture specification* is used in the *processor behavior analysis*. While our approach has been investigated only w.r.t. sequential processors, these settings allow a similar modular integration of other hardware components, e.g., pipeline specification or multicore processors with FPU, as well as of other resource analyses, e.g., power consumption or memory usage. However, we consider fixed the analysis method, i.e., abstract interpretation.

In this paper we present a methodology for the modular integration of abstractions for resource analysis over a trusted core - the concrete semantics specification of a system. We instantiate this methodology on analyses for WCET [28] (e.g., control flow analyses such as constant propagation [8] and interval analysis [22], and processor behavior analyses, namely, may and must for data and instruction caches [25]). Our contribution focusses more on the engineering aspects of integrating two formal methods: the concrete semantics of systems (which are formally specified and tested to form a trusted core) and the abstract semantics for resource analyses (the resource exemplified here is the execution time). The integration of these two formal methods is mutually building strength to both of them. Namely, making the abstraction to draw its abstract operators directly from the concrete ones available in the trusted core builds confidence that the analysis is applied to a model which is faithful to the system of origin. In turns, the system specified by a trusted core, besides being tested w.r.t. the accuracy of the semantics specification, is now also guaranteed to work correctly w.r.t. more subtle properties, e.g., tight and safe resource bounds.

Our tactics to integrate abstractions over the concrete semantics specification starts by identifying a set of semantic entities used to "cut" slices through the concrete execution. Then, the abstraction-specific semantic entities are combined with the concrete slices, in a modular fashion. A more intuitive view of the tactics used for abstractions is given in Fig. 1. The system specification, in Fig. 1

(left), is described structurally by a configuration *Config*, and semantically by a set of rewrite rules of the form $X \Rightarrow X'$. The underlying modularity of the specification, in Fig. 1 (middle), is captured with the possibly overlapping sub-configurations (i.e., $C_1$ and $C_2$) of *Config*. A sub-configuration identifies a part of the system with it afferent functionality (e.g., since $X, X'$ are subterms of $C_1$, the rule $X \Rightarrow X'$ implements the functionality of $C_1$). An operator "?" (also called guard) specifies how these parts communicate between them. i.e., $?(X' \cap Y)$ means that $Y \Rightarrow Y'$ (in $C_2$) is applied after $X \Rightarrow X'$ (in $C_1$). If we see the specification as a program, then "?" is an `assert (cond)` statement which allows the execution to proceed if `cond` is true. The abstraction integration, in Fig. 1 (right), is done in two steps. First, an abstract configuration, *CA*, encapsulates the necessary information from the existing sub-configurations (e.g., the parts of $C_1$ and $C_2$ used by $X$, $X'$ and respectively $Z$, $Z'$) and isolates, in this way, the important rules (e.g., the rules using the # operator). Second, the abstraction specify how to control the system at the points of interest. For example, the two conditions in **Abstraction** enable the following execution steps (represented with the dashed lines): rule $Z \Rightarrow Z'$, the guard/assert statement $?(\#Z' \cap \#X)$ and rule $X \Rightarrow X'$. The second guard $?(\#X' \cap \#Y)$ fails and this execution stops. **Abstraction** has the mechanism (i.e., rewrite rules) to perform such executions, to collect and to process their results.

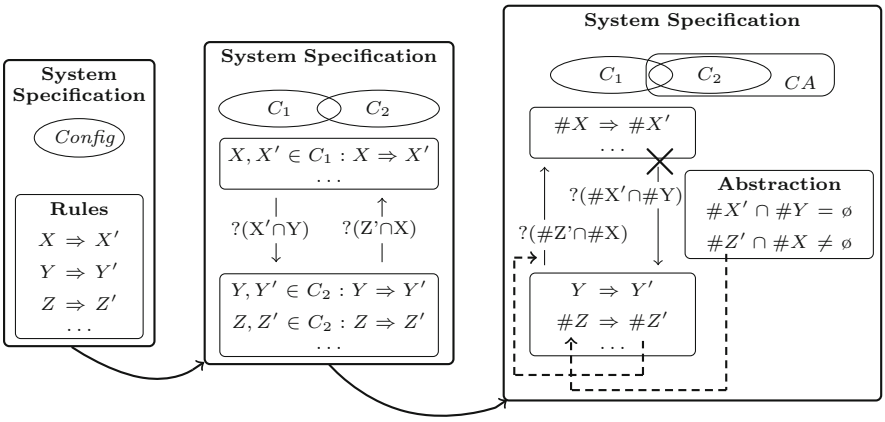


**Fig. 1.** Workflow for modular resource analysis: in the initial specification (left), we identify the components and the inter-component communication (middle), in order to apply abstractions over an existing (part of the) infrastructure (right).

The formal and executable features of the semantics are provided by a specialized framework, called $\mathbb{K}$ [24] which emerged from the rewriting logic [21]. In the graph theory $\kappa$ represents the connectivity of a graph. Likewise, $\mathbb{K}$ aims to connect different semantics paradigms such as (modular) small/big step, continuation, or reduction semantics distilled into a cell-based notation á la Chemical

Abstract Machine. $\mathbb{K}$ integrates the main features of all these semantics into the algebraic environment of rewriting logic which sets solid foundations, i.e., the trusted core, for automated verification. Our modular resource analysis is prototyped in $\mathbb{K}$-Maude [10], the implementation of $\mathbb{K}$ on top of the rewriting logic tool Maude [7]. The experimental results are presented with respect to a reusability metric which measures the implementation from the following perspective: keeping the formal semantics, how much of the concrete system is used, while the abstractions are applied. Hence, with this metric we asses how much of the effort spent in specifying the concrete system is actually used by abstraction. We select and conduct experiments on a subset of the Mälardalen benchmarks [13].

**Related Work.** There are several works concerning formal development of WCET analysis workflows. We mention here the general resource-driven methodology advocated by the Hume project [14], the theorem proving-based framework, via Coq [5], which aims towards the WCET-certifiable compilation and the use of symbolic execution to prove the bounds, while tightening them [4].

The Hume project uses a domain-specific, multi-level (low- to higher-order) language called Hume [14]. Hume combines concurrent finite-state automata (called boxes) as the specification mechanism with the executability support of the functional programming (i.e., through pattern-matching and rich typing information). Both Hume and our approach share similar principles w.r.t. modularity. To a Hume box it corresponds a $\mathbb{K}$ module which is a rewriting-logic theory (through its compilation into a Maude module). To a Hume wiring (connection between boxes) it corresponds communication tokens between $\mathbb{K}$ modules. One key difference is how the abstractions are handled: Hume embeds the abstractions in the workflow, while our methodology advocates for a more flexible scheme, lifting the definition of abstractions at the application level.

The work in [5] uses the infrastructure of the CompCert [17] verified C compiler to formally prove a specific control-flow analysis - the loop bound estimation. As in our methodology for modular integration of abstractions, the Coq formalization of a loop bound analysis is also integrated into an existing workflow (i.e., of CompCert). The working language is the CompCert RTL intermediate representation which is augmented with loop scoping in order to extract control-flow information from loops. While we share a similar low-level language representation, our methodology does not explicitly extend the existing concrete configuration, but it slices it using the wrapping. Another difference is that the Coq formalization of the loop-bound abstraction is fully proved within the workflow, while our approach relies on offline proving of the abstractions.

The WCET analysis workflow is wrapped into a counterexample guided abstraction refinement cycle, in [4]. The point is to consider a WCET bound and to further tighten/prove it using a symbolic execution engine. Both our methodology and [4] follow the same ILP + AI method, w.r.t. the infrastructure for WCET analysis. The key difference is in the workflow, the symbolic execution approach is formal w.r.t. the results of the WCET analysis, while our methodology also proves that the computation for the WCET bound is correct

(i.e., up to the specification). The combined ILP + AI approach for WCET analysis was straightforwardly encoded in [2]. In this paper we propose a general methodology to integrate abstract executions and show how our previous encoding of ILP + AI is just an instance of this methodology. Moreover, we present the implementation and experiments w.r.t. the quality of integration.

While the general approach to evaluate abstractions is through experimentation, several techniques address this problem from the more systematic perspective - that of a metric definition. The flow is simple: define a standard (i.e., a set of characteristics) and project each abstraction on this standard. For [12], the abstraction-refinement procedures are compared w.r.t. a standard counterexample. In other words, the metric evaluates the capabilities (i.e., accuracy, performance) of an abstraction. Along the same lines, the standard (i.e., a collection of metrics) defined in [23] captures predictability aspects of cache memories w.r.t. the impact on the WCET analysis. Our metric does not measure the accuracy of an abstraction, as in [12,23]. We propose as standard, the structure of the concrete semantics, hence an abstraction is measured from a more structural point of view. It is more closely related to the metrics in software engineering [11] which evaluate integration of design patterns (in fact our integration of abstraction could be assimilated with a design pattern).

**Paper Outline.** This paper is organized as follows: Sect. 2 covers some background notions of $\mathbb{K}$ and overviews some aspects of the abstract interpretation and its application to WCET analysis. Sections 3 and 4 overview the general system design and, respectively, its instance for the timing analysis. Section 5 covers some implementation and experimental details with the current prototype while Sect. 6 contains the conclusions.

## 2 Preliminaries

**The $\mathbb{K}$ framework:** We introduce $\mathbb{K}$ and illustrate some key concepts in our approach by using a simple specification of an embedded application running on a very basic architecture.

*Example 1.* Without restricting the generality, we consider a toy system given by: (1) a RISC assembly language as the programming language of choice which consists of representatives of the arithmetic-logic instructions - `add`, the branch and jump instructions - `beq`, and the memory-access instructions - `lw` and `sw`; (2) the architecture which features a minimal direct-mapped data cache (with one data assigned per cache line) and a main memory module. □

Next, we introduce the development platform - the $\mathbb{K}$ framework - which is a specialization of rewriting logic for the specification and analysis of programming languages. A $\mathbb{K}$ specification consists of *configurations*, *rules*, and *computations*. A *configuration* defines all the semantic entities necessary for representing the system (or program) states. Namely, the $\mathbb{K}$ configuration is a nested set of $\mathbb{K}$-cells where the $\mathbb{K}$-cell is essentially a generic type formed by a *label* giving the identity of the cell, and the sort of the cell's contents, defined as $\langle\ \mathit{ContentSort}\ \rangle_{\mathsf{label}}$.

*Example 2.* Our toy RISC assembly language configuration, $Cfg_{toyRISC}$, is:

$$Cfg_{toyRISC} \equiv \langle K \rangle_{\mathsf{k}}\ \langle Reg \rangle_{\mathsf{pc}}\ \langle Reg \mapsto Val \rangle_{\mathsf{regs}}$$

where k, pc, and regs are $\mathbb{K}$-cells, and *Reg* and *Val* are sorts for registers and respectively stored values. The cell called k is specially designated in $\mathbb{K}$ to contain the list of computational tasks (of sort $K$), according to the continuation-based semantics. Hence, the cell k is supposed to be the functional engine of the $\mathbb{K}$-specification. The cell pc holds the program counter register while the cell regs maintains the integer register file - a map from registers to values. In a similar fashion, the configurations for the data cache memory - $Cfg_{ToyDC}$ - and the main memory - $Cfg_{toyMM}$ - are as follows:

$$Cfg_{ToyDC} \equiv \langle K \rangle_{\mathsf{k}}\ \langle CAddr \mapsto Val \rangle_{\mathsf{dc}}$$

where dc is the content mapping cache addresses *CAddr* to data *Val* and

$$Cfg_{toyMM} \equiv \langle K \rangle_{\mathsf{k}}\ \langle MAddr \mapsto Instr \rangle_{\mathsf{cmem}} \langle MAddr \mapsto Val \rangle_{\mathsf{dmem}}$$

where the $\mathbb{K}$-cells cmem and dmem are the code and respectively data memory mapping addresses *MAddr* to instructions *Instr* and, respectively, to data *Val*. □

The *rules* in $\mathbb{K}$ describe patterns of system execution which produce changes in the states of the system (i.e., in the instantiations of the configuration). For example, the rules over subterms of $Cfg_{toyRISC}$ give semantics to the language constructs (i.e., the instructions `add`, `beq`, `lw` and `sw` of our toy RISC language) while the rules over $Cfg_{ToyDC}$ or $Cfg_{toyMM}$ give semantics to the architecture (e.g., cache hit/miss on read/write requests and main memory read/write operations). The rules are classified as: *computational rules*, which may be interpreted as transitions in a program execution, and *structural rules* that modify a term to enable the application of a computational rule. Unless labeled with the keyword `[structural]`, a $\mathbb{K}$-rule is computational.

*Example 3.* The semantics of the load instruction - `lw Rd, Off(Rs);` updates a destination register `Rd` with the value at the memory address calculated based on a source register value `Rs` and an offset `Off`. The following $\mathbb{K}$ rules encode the semantics of `lw` - rule `R_LW` and of register update - rule `R_RU`:

$$\langle\ \underbrace{\texttt{lw}\ Rd, Off(V_1);}_{\texttt{updReg}(\boxed{\texttt{getd}(V_1 + Off)}, Rd)}\ \rangle_{\mathsf{k}} \qquad [\texttt{R\_LW}, \text{structural}]$$

$$\langle \underbrace{\texttt{updReg}(V, R)}_{\cdot} \rangle_{\mathsf{k}}\ \langle \cdots\ R \mapsto \underbrace{\_}_{V}\ \cdots \rangle_{\mathsf{regs}} \qquad [\texttt{R\_RU}]$$

A $\mathbb{K}$ rule uses a specialized bi-dimensional notation to specify the *rewriting context* and the location of the rewriting. For example, in the rule `R_LW`, the `lw` instruction appearing inside cell k is transformed into a register update, via a `getd` data request from the main memory. Note that in rule `R_LW` the context is given by the cell k. Moreover, the boxing of the term $\texttt{getd}(V_1 + Off)$

shows that over this term we apply the reduction semantics. Namely, when a term $T$ is boxed at the top of $\mathsf{k}$ cell in the context $C$, i.e., $\langle C\boxed{T} \ \cdots\rangle_{\mathsf{k}}$, then $\mathbb{K}$ applies the reduction semantics style, by pushing $T$ - called redex - outside the box, at the top of the continuation, i.e., $\langle T \curvearrowright C\boxed{\ } \ \cdots\rangle_{\mathsf{k}}$. Hence, the reduction-based feature of $\mathbb{K}$ pushes $\mathtt{getd}(V_1 + Off)$ at the top of the $\mathsf{k}$ cell, i.e., $\langle \mathtt{getd}(V_1 + Off) \curvearrowright \mathtt{updReg}(\boxed{\ }, Rd) \ \cdots\rangle_{\mathsf{k}}$. After the reduction of $\mathtt{getd}(V_1 + Off)$ to its normal form, an integer in this case, the result is pushed back into the box, i.e., $\langle \mathtt{updReg}(\boxed{V}, Rd) \ \cdots\rangle_{\mathsf{k}}$, then the box is dissolved leaving $\mathtt{updReg}(V, Rd)$ at the top of $\mathsf{k}$.

The rule R_RU can be applied after the rule R_LW, which places the updReg operation on top of cell $\mathsf{k}$. The result is twofold: in $\mathsf{k}$, the new computation is the empty task, represented by "$\cdot$" notation standing for the *void element*, and *somewhere* in the regs cell *any* previous value associated to $R$ is replaced by $V$. Note that the $\mathbb{K}$ notation for *somewhere* is given by the ellipses "$\cdots$" appearing near the walls of cell regs (i.e., the ellipses stand for *other elements in the respective cell*, the "$\cdot$" included). Also, note that *any* value (contained in the register $R$) is denoted by the wildcard "_" (i.e., $R \mapsto \_$). □

A *computation* in $\mathbb{K}$ is a sequence of rewrite rules applications.

*Example 4.* Let us consider yet another rule for our toy specification which captures a data request from a given memory address *Addr*:

$$\underbrace{\langle \mathtt{getd}(Addr) \ \cdots\rangle_{\mathsf{k}}}_{\mathtt{retd}(V)} \ \langle \cdots \ Addr \mapsto V \ \cdots\rangle_{\mathsf{dmem}} \quad [\mathtt{R\_MD}]$$

Note that this rule, R_MD, is working on the configuration $Cfg_{toyMM}$, according to the cells used by this rule. Note also that while R_RU has to be part of $Cfg_{toyRISC}$, for R_LW we have no restrictions but we prefer to place it in $Cfg_{toyRISC}$ as well. We then observe that the sequence of rule applications R_LW, R_MD, R_RU, intercalated with the reduction mechanism (denoted as $\sqcap$ for pushing the redex out of the box, and $\sqcup$ for pushing it inside the box), is a $\mathbb{K}$-computation. Namely, the evolution of the continuation - the contents of the $\mathsf{k}$ cell - for the substitution $Rd/5, Off/1, \mathtt{V}_1/3, Addr/4$ and $V/7$, is:

$$\langle \mathtt{lw}\ 5, 1(3);\rangle_{\mathsf{k}} \overset{\mathtt{R\_LW}}{\rightarrow} \langle \mathtt{updReg}(\boxed{\mathtt{getd}(3+1)}, 5)\rangle_{\mathsf{k}} \overset{\sqcap}{\rightarrow} \langle \mathtt{getd}(\boxed{3+1}) \curvearrowright \mathtt{updReg}(\boxed{\ }, 5)\rangle_{\mathsf{k}}$$
$$\overset{\sqcap}{\rightarrow} \langle 3+1 \curvearrowright \mathtt{getd}(\boxed{\ }) \curvearrowright \mathtt{updReg}(\boxed{\ }, 5)\rangle_{\mathsf{k}} \overset{\mathtt{R+}}{\rightarrow} \langle 4 \curvearrowright \mathtt{getd}(\boxed{\ }) \curvearrowright \mathtt{updReg}(\boxed{\ }, 5)\rangle_{\mathsf{k}}$$
$$\overset{\sqcup}{\rightarrow} \langle \mathtt{getd}(4) \curvearrowright \mathtt{updReg}(\boxed{\ }, 5)\rangle_{\mathsf{k}} \ \langle \cdots \ 4 \mapsto 7 \ \cdots\rangle_{\mathsf{dmem}} \overset{\mathtt{R\_MD}}{\rightarrow} \langle \mathtt{retd}(7) \curvearrowright \mathtt{updReg}(\boxed{\ }, 5)\rangle_{\mathsf{k}}$$
$$\overset{\mathtt{R\_RD}}{\rightarrow} \langle 7 \curvearrowright \mathtt{updReg}(\boxed{\ }, 5)\rangle_{\mathsf{k}} \overset{\sqcup}{\rightarrow} \langle \mathtt{updReg}(7, 5)\rangle_{\mathsf{k}} \ \langle \cdots \ 5 \mapsto 2 \ \cdots\rangle_{\mathsf{regs}} \overset{\mathtt{R\_RU}}{\rightarrow} \langle \cdot\rangle_{\mathsf{k}} \ \langle \cdots \ 5 \mapsto 7 \ \cdots\rangle_{\mathsf{regs}},$$

where the rule R_RD simply rewrites $\mathtt{retd}(V)$ into $V$ if $V$ is an integer (i.e., a basic value in *Val*). Hence, this computation starts from the module corresponding to $Cfg_{toyRISC}$, via R_LW, and goes to $Cfg_{toyMM}$ to execute R_MD, and it ends in $Cfg_{toyRISC}$ with R_MD. □

In the settings of *Example* 1, the rule R_MD specifies part of the main memory behavior. This rule emphasizes two key features of our approach. First, the configuration $Cfg_{toyMM}$ of the main memory has also a code memory cell (i.e., cmem in *Example* 2) which is not used in the rule R_MD. The reason is the *configuration abstraction* mechanism of $\mathbb{K}$, which allows compact representations of

the rewrite rules (and hides a rule completion mechanism). The intuition is to use only the relevant semantic entities when writing a particular rule. Second, the `getd` request is placed on top of cell `k` after applying the rule `R_LW`, as seen in *Example* 3, and it enables the application of the rule `R_MD`. At its turn, `R_MD` places `retd` at the top of communication to signal the enabling of `R_RU`. Hence, we specify an inter-component communication via `getd` and `retd`.

**Abstract Interpretation:** Since it was introduced in [9], abstract interpretation established itself as one of the major program reasoning techniques, along with model checking and deductive verification. For a given programming language, abstract interpretation is used to systematically design abstract semantics which target the analysis of specific properties. We introduce the settings of abstract semantics for WCET analysis, as in [28].

A program analysis relies on the following two elements: an abstract domain and an abstract semantics. The abstract domain is defined by a complete semilattice, a pair of monotonic (with respect to partial orderings, both in concrete and abstract) functions - called representation and concretization. The representation function maps/represents concrete to abstract states while the concretization function maps abstract states to sets of concrete states. The following translation holds: a concrete state is represented by an abstract state which is concretized to a set of states containing the particular concrete state. A third function, called abstraction, is defined in terms of the representation function and, together with the concretization function, forms a Galois connection. An essential part is the abstract semantics which re-implements the transfer functions of the concrete semantics, under certain requirements. The abstract semantics is used to solve the program analysis problem, by employing a fixpoint computation.

In the context of our approach for modular integration of abstractions, and according to [28], we consider abstract interpretation-based analyses both at the level of the program (i.e., value analyses), and architecture (i.e., cache analyses).

## 3 A General System Design

Let us consider a complex computational system, specified in $\mathbb{K}$, that has the configuration $C_{global}$ which comprises *all* semantic entities that are necessary to capture the system behavior. We take the $C_{global}$ configuration and split it into a number of (not necessarily disjoint) sub-configurations: $C_1, C_2, \ldots, C_n$. Informally, each sub-configuration handles a well-defined component of the system. The configuration splitting induces a first stage of modularization of the semantics. As such, we distinguish two types of modules *functional* and *structural*, according to the presence or, respectively, the absence of the `k` cell in the sub-configuration associated with the module. Given that cell `k` handles the sequencing of the computations, the (functional) modules containing this cell are to be responsible of producing slices of computations.

*Example 5.* For our simple specification, introduced in *Example* 1, the $C_{global}$ is:

$$\langle K \rangle_{\mathsf{k}}\, \langle Reg \rangle_{\mathsf{pc}}\, \langle Reg \mapsto Val \rangle_{\mathsf{regs}}\, \langle CAddr \mapsto Val \rangle_{\mathsf{dc}}\, \langle MAddr \mapsto Val \rangle_{\mathsf{cmem}}\, \langle MAddr \mapsto Val \rangle_{\mathsf{dmem}}$$

while the modularization consists of the sub-configurations $Cfg_{toyRISC}$, $Cfg_{ToyDC}$, and $Cfg_{toyMM}$, each inducing a functional module. Moreover, the specification is equipped with a structural module, which contains the arithmetic (i.e., the rules for the `add`, `lw`, and `sw` instructions) and logic operations (i.e., for the `beq` instruction) on 32-bits. □

Apart from its computational purpose, the cell `k` also facilitates the inter-modular communication through tokens (e.g., `getd` is the token for data request from the main memory as seen in *Example* 4). Structurally, a token is part of the module interface and semantically, it is a computational task which acts like a guarded message passing between modules. Therefore, any execution in this system is an interleaving of inner-module computations and guarded messages that are exchanged between the modules. We assume that any token is sent and received by one module, however this model is amenable to extensions to accommodate concurrent exchanges of tokens.

The configuration abstraction of $\mathbb{K}$ completes the partially defined rules with the cells which were omitted. The mechanism of rule completion identifies first the nesting structure of the configuration, then it inserts the missing cells in the correct place, defining like this the notion of *rule context*. As such, it heavily relies on the shape defined by the configuration. Our idea is to integrate abstractions by changing the configuration shape, through $\mathbb{K}$ cell wrapping, creating in this way new rule contexts.

*Example 6.* The rule `R_MD` (i.e., in *Example* 4) is completed according to $Cfg_{toyMM}$ as:

$$\frac{\langle \texttt{getd}(Addr) \;\cdots\rangle_{\mathsf{k}}\; \langle \cdots\; Addr \mapsto V \;\cdots\rangle_{\mathsf{dmem}}\; \langle C \rangle_{\mathsf{cmem}}}{\texttt{retd}(V)} \qquad [\texttt{R\_MD\_completed}]$$

where the code memory content $C$, in `cmem` cell, is unchanged. However, if the cells `k` and `dmem` are wrapped in a new cell then the rule completion does not add `cmem`. Hence, `getd` acts like a guarded message, the guard being here the structure of $Cfg_{toyMM}$. Moreover, we can use also other types of guarded messages by giving the message rule as a conditional rule where the guard is the predicate in the condition. Note that the conditional rules are denoted in $\mathbb{K}$ by the keyword "when", i.e., "$l \Rightarrow r$ when $p$" where $l$ and $r$ are the left- and, respectively, the right-part of the rewrite while $p$ is the predicate denoting the condition which has to be met in order for the rewrite to be triggered. For example, such a guarded message is `retd` and the conditional rule associated to it is denoted as: $\frac{\text{"}\texttt{retd}(V)}{V}$ when $\texttt{isInt}(V) =_{Bool} true$ [`R_RD`]". □

We present the abstraction integration methodology as a meta-algorithm, in Fig. 2, then we instantiate it for the WCET analysis workflow. The integration considers a set of modules $M_i$, each being represented by sub-configurations

**Input:**

$M_i$ - $\mathbb{K}$ modules with corresponding sub-configurations $C_i$, with $i = 1..n$;
$T = \{t_{i,j} \mid i, j = 1..n, i \neq j\}$ - tokens between the modules $M_i$ and $M_j$;
$\langle\langle\cdot\rangle_{\mathsf{k}}\ \langle S_0\rangle_{\mathsf{states}}\langle RE\rangle_{\mathsf{slicePattern}}\ Rest\rangle_{\mathsf{abst}}\ \langle\cdot\rangle_{\mathsf{concr}}$ - the abstract configuration inside cell abst where $S_0$ is the initial abstract state, $RE$ the slicing pattern given as, e.g., a regular expression over $T$, and *Rest* being other abstraction-specific information, and an empty concrete configuration inside the concr cell.

**Output:**

$\langle\langle S\rangle_{\mathsf{states}}\ \langle RE\rangle_{\mathsf{slicePattern}}\ Rest\rangle_{\mathsf{abst}}$ - the final state $S$ of the abstraction, identified by the absence of the abstract computation cell k inside cell abst.

INIT

$$\langle\langle \underbrace{t_{i,j} \curvearrowright t_{k,l}}_{\mathrm{wait}(t_{k,l})} \cdots\rangle_{\mathsf{k}}\ \langle S\rangle_{\mathsf{states}}\ \cdots\rangle_{\mathsf{abst}}\quad \langle \underbrace{\cdot}_{\langle t_{i,j}\rangle_{\mathsf{k}}\ \langle \flat(S)\rangle_{\mathsf{state}}} \rangle_{\mathsf{concr}}$$

STOP

$$\langle\langle \underbrace{\mathrm{wait}(t_{k,l})}_{\mathrm{process}_{k,l}(c \sqcup S) \curvearrowright t_{k,l}} \cdots\rangle_{\mathsf{k}}\ \langle S\rangle_{\mathsf{states}}\ \cdots\rangle_{\mathsf{abst}}\quad \underbrace{\langle\langle t_{k,l} \cdots\rangle_{\mathsf{k}}\langle \sharp(c)\rangle_{\mathsf{state}}\rangle_{\mathsf{concr}}}_{\cdot}$$

RESUME

$$\langle\underbrace{\langle \mathrm{processed}_{k,l}(S') \cdots\rangle_{\mathsf{k}}}_{\cdot}\ \langle S\rangle_{\mathsf{states}}\ \cdots\rangle_{\mathsf{abst}}\ \underbrace{\langle\cdot\rangle_{\mathsf{concr}}}_{\cdot} \qquad \text{when} \quad S' < S$$

$$\langle\underbrace{\langle \mathrm{processed}_{k,l}(S') \cdots\rangle_{\mathsf{k}}}_{\cdot}\ \langle \underbrace{S}_{S'} \rangle_{\mathsf{states}}\ \cdots\rangle_{\mathsf{abst}} \qquad \text{when} \quad S' \not< S$$

REPEAT

$$\langle\langle \underbrace{\cdot}_{RE} \rangle_{\mathsf{k}}\ \langle RE\rangle_{\mathsf{slicePattern}}\ \cdots\rangle_{\mathsf{abst}}$$

**Fig. 2.** The meta-algorithm for modular integration of abstractions.

wrapped into the abstract configuration. $M_i$s use the token sets $t_{j,k}$ to communicate, via the cell k. The structural modules are not part of the set of $M_i$s, but their rules can be applied wherever necessary (e.g., the rule for integer addition R+ used in the computation from *Example* 4).

This meta-algorithm identifies the following stages: INIT, STOP, RESUME, and REPEAT and it uses $\langle RE\rangle_{\mathsf{slicePattern}}$ as the regular pattern of slicing the concrete executions. The pattern $RE$ is repeatedly pushed in the abstract continuation via the rule REPEAT. As such, the flow of the rules in the meta-algorithm is (REPEAT.(INIT.STOP.RESUME)$^*$)$^*$. Namely, the sequence INIT.STOP.RESUME is repeatedly executed until the abstraction reaches the fixpoint (the first rule of RESUME) or the slicing pattern $RE$ is consumed. If, however, $RE$ is consumed without reaching the fixpoint, i.e., the abstract k cell is empty, then the rule REPEAT is applied so (INIT.STOP.RESUME) is triggered again.

The connection INIT.STOP is made at the top of the abstract k cell via "wait($t_{k,l}$)" which means that the abstract computation is waiting for the concrete computation triggered by INIT to reach the stage where the token $t_{k,l}$ is computed in concrete. We recall that these tokens are functioning as execution guards which, based on the evaluation of the concrete state (the contents of cell

state), let the concrete execution pass to another module. However, $\flat(S)$, the concretization of the abstract state $S$, produces a wrapping around the concrete state - $\sharp(c)$ - which does not allow the triggering of the concrete rule via the context abstraction mechanism. Hence, the concrete execution is cut at $t_{k,l}$ and the rule STOP is triggered. Also, we assume that "process$_{k,l}$"- which processes the assimilation of the newly discovered concrete state $c$ into the abstract state $S$ - reaches the normal form "processed$_{k,l}$". This assumption ensures the connection STOP.RESUME. Finally, the eventual repetition of the sequence, i.e., the connection RESUME.INIT, is produced by the second rule in RESUME which consumes the "processed$_{k,l}(S')$" top of the continuation and clears the way for a new application of the INIT rule. Note that we make use of a mechanism of unfolding the regular expression $RE$ into computation tasks $t \curvearrowright t'$ but we do not insist over this aspect here.

*Example 7.* Let us assume that we integrate a *value analysis* over the specification described in *Example* 1. This analysis returns the sign of the values referred by load/store instructions, where the sign lattice is $\{?, +, -, 0, 0+, -0, -0+\}$. The meta-algorithm takes the modules $M_1$ - $Cfg_{toyRISC}$, $M_2$ - $Cfg_{ToyDC}$, $M_3$ - a part of $Cfg_{toyMM}$ (i.e., only k and data memory dmem cells because the analysis focuses on data). As observed in *Example* 6, the token $t_{1,3}$ is getd, used as a memory request for a cache miss on lw, while $t_{3,1}$ is retd. Note that $t_{1,2}$ also contains tokens used for the store instruction sw but, for brevity, we decide not include them here.

When we integrate the value analysis, we wrap in the concr cell only a part of $M_3$, i.e., the cells k and dmem. In this way, we isolate the necessary behavior to capture the accesses to the data memory. In other words, the wrapping in concr slices the main memory specification. The abstraction-specific semantic entities are wrapped in the abst cell in the *Rest* part which, for this value analysis, contains a representation of the control-flow graph (CFG). Note that CFG is extracted using a similar integration of an abstraction for CFG extraction into the meta-algorithm.

Now, if the slicing pattern $RE$ unfolds into $\langle\langle \texttt{getd}(A) \curvearrowright \texttt{retd}(V) \curvearrowright \text{CFG}\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{abst}}$ then the INIT rule with the abstract state $\langle\cdots\ \langle\cdots\ 4 \mapsto ?\ \cdots\rangle_{\mathsf{states}}\ \cdots\rangle_{\mathsf{abst}}$ pushes into the concrete cell $\langle\langle \texttt{getd}(4)\rangle_{\mathsf{k}}\langle\cdots\ 4 \mapsto \sharp(7)\ \cdots\rangle_{\mathsf{dmem}}\rangle_{\mathsf{concr}}$. The concrete computation stops with $\texttt{retd}(\sharp(7))$ at the top of the k cell, since rule R_RD cannot be triggered due to the fact that the rule condition $\texttt{isInt}(\sharp(7))$ is not *true*. Then, the rule STOP is enabled, $\sharp(7)$ goes into the abstract continuation as $\langle\cdots\ \langle\text{process}(\sharp(7) \sqcup (4 \mapsto ?))\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{abst}}$. Next, process$(\sharp(7) \sqcup (4 \mapsto ?))$ reaches the normal form "processed$(4 \mapsto +)$)" and the second rule in RESUME is triggered. Note that the slicing pattern $RE$ depends on the CFG, so the rule REPEAT will trigger the fixpoint computation for the entire program abstracted into the CFG containing only loads and stores. Finally, we remark that we show here only the part of $RE$ for loads and we omit the part for stores. □

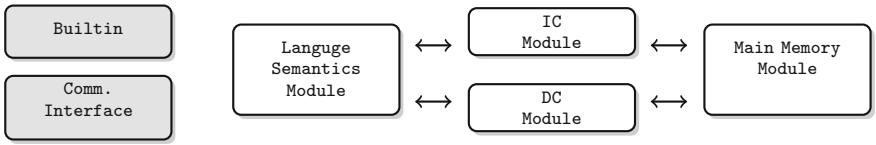


**Fig. 3.** The modular organization of our WCET analysis workflow with structural (i.e., support operations and communication interface) and functional modules (i.e., language semantics, cache and main memories).

## 4 Modular Timing Analysis

**The System:** Our proposed system consists of the formal executable semantics of the MIPS IV assembly language (which plays the role of the processor), parametric specifications of instruction and data caches, of the main memory, as well as specialized support modules. The parametrization refers to both the cache structure (e.g., size, associativity) and functionality (e.g., replacement policies, writing policies).

The overall system, shown in Fig. 3, consists of several $\mathbb{K}$ modules. Next we elaborate the structural modules Builtin and Interface as well as the functional modules Language Semantics, IC and Main Memory. For each of these modules, we discuss the corresponding concrete configurations, which from our methodology perspective described in Fig. 2, are wrapped into the concr cell. The complete configuration, containing both concr and abst, is described in Sect. 4.2 - for the data analyses and Sect. 4.3 - for the ILP + AI method.

*Language Semantics* - We encode the MIPS IV assembly language as supported by the Simplescalar toolset. Its complete configuration $Cfg_{Lang}$ is given as an extension of $Cfg_{toyRISC}$ from Example 2:

$$Cfg_{Lang} \equiv Cfg_{toyRISC}\langle Reg\rangle_{\mathsf{lo}}\langle Reg\rangle_{\mathsf{hi}}\langle Reg\rangle_{\mathsf{ra}}\langle Val\rangle_{\mathsf{fcc}}\langle Val\rangle_{\mathsf{break}}\langle FReg \mapsto Val\rangle_{\mathsf{fregs}}$$

The concrete language configuration has the cells k, pc and regs from $Cfg_{toyRISC}$, the floating-point register file, fregs, and also two flags, break and fcc (for abrupt termination of execution and respectively a floating point parameter). It also features several specialized registers for the multiplication/division results, lo and hi, and the return address of a function call, ra. Each $\mathbb{K}$ cell displays sorting information.

*Main Memory* - We emulate the organization of an assembly file into code and data sections, represented in the cmem cell and respectively dmem cell in *Example* 2. The concrete main memory configuration, $Cfg_{toyMM}$, plays an important role in both control and data abstractions because it stores the input program.

*IC Memory* - The instruction cache configuration, $Cfg_{IC}$, captures both structural and functional aspects of a cache memory:

$$Cfg_{IC} \equiv Cfg_{toyIC}\langle Addr \mapsto Val\rangle_{\mathsf{ages}}\langle Addr\rangle_{\mathsf{repl}}\langle Prm \mapsto Val\rangle_{\mathsf{prm}}\langle Val\rangle_{\mathsf{ilen}}\langle Addr\rangle_{\mathsf{faddr}}\langle Type \mapsto Val\rangle_{\mathsf{profl}}$$

The k cell and instruction content ic cell are inherited from the $Cfg_{ToyIC}$ configuration. The ages cell contains, for each cache line, the associated age information of its content. The repl cell has the address of the cache block as the next eviction candidate (computed based on the age information). The profl and prm cells store the hit/miss counts and the cache parameters (e.g., size, associativity), while the ilen and faddr are the instruction length and the address of the first instruction in the program, respectively. The data cache configuration of the DC Module (not presented) is an extension of $Cfg_{IC}$ with new cells to capture parametric modeling for writing policies (i.e., FIFO or LRU).

*Builtin* - It is a structural module containing definitions for semantic operations on 16- and 32-bit signed and unsigned integers, and single and double precision for floats.

*Interface* - This structural module contains the communication tokens, $t_{i,j}$s, between two arbitrary modules $M_i$ and $M_j$, as described in Fig. 2.

This system is designed to simulate the execution of a program on an underlying architecture. We call this the *concrete specification* of the system. The execution of one instruction consists of a sequence of code and data requests to the memory system described by several $\mathbb{K}$ modules (IC Module, DC Module, and Main Memory in Fig. 3).

The modularity of our design allows us to define abstractions directly over the existing modules while these are kept unmodified. The abstract configuration has two components: (a) a definition-based set of $\mathbb{K}$ cells, derived from the concrete definition of the system and (b) an abstraction-specific set of $\mathbb{K}$ cells, to represent abstract datatypes and other necessary auxiliary constructs. We elaborate next on how we encode four abstractions: constant propagation and the interval analysis in Sect. 4.2, and the ILP+AI combined method in Sect. 4.3.

**The Data Abstractions:** The constant propagation and the interval analysis are widely used static analyses for WCET analysis, according to [28]. In this section we discuss how these two analyses are integrated within our methodology.

A constant propagation analysis produces, at each program point, the set of variables (i.e., register values) having constant values. According to the general scheme for an abstract interpretation based analysis, from Sect. 2.2, we need to define the abstract domain and abstract versions of the language operations. The unknown value of a variable is abstractly represented by a special symbolic value. Also, the abstract operations extend the concrete ones (in Builtin Module) with the unknown value case.

To encode the constant propagation we retain, from the concrete system, the language registers and the code memory, wrapped in the concr cell. The abstraction-specific component, abst, is empty. Therefore, the configuration for the constant propagation, restricted to the integer domain, is the following:

$$\langle \langle Reg \mapsto\ Val \rangle_{\mathsf{regs}} \langle\, Val \rangle_{\mathsf{break}} \langle\, Val \rangle_{\mathsf{lo}} \langle\, Val \rangle_{\mathsf{hi}} \langle Addr \mapsto\ Val \rangle_{\mathsf{cmem}} \rangle_{\mathsf{concr}}$$

The interval analysis from [22] relies on an abstract representation of a language value as an interval. Our assembly language uses several types of values: integers and floats, and the interval-based values applies to all of them. Next,

we refer to the interval analysis encoding on the integer domain. We follow the same design steps as before: we start with the abstract domain and define new sort information to represent the interval bounds. This integration is seamless with respect to the semantics rules in language semantics module, however a new `Builtin` is required. Once the abstract domain and the `Builtin` module are defined, we proceed to integrate the interval analysis in the same spirit. Since this analysis generalizes the constant propagation, the following relation exists between $Cfg_{CP}$ and $Cfg_{IA}$ (defined below). The abst cell content of $Cfg_{CP}$ remains the same (modulo interval-related sorting) and we add an abstraction-specific cell, called mdop. The configuration for the interval analysis is defined as: $Cfg_{IA} \equiv \langle Cfg_{CP} \langle Val \rangle_{\mathsf{mdop}} \rangle_{\mathsf{abst}}$. The cell mdop is necessary because the multiplication and division operations require two special registers hi and lo to store the result (for the multiplication) or the quotient and the remainder (for the division). After such arithmetic operation, the interval analysis suffers due to rounding errors for lower and upper bounds of the interval result. For example, when multiplying two intervals, we temporarily store the result in mdop before we transfer it into the hi and lo from concr.

**The ILP+AI Approach:** A successful approach for WCET analysis proposes (1) an integer linear programming (ILP) solution, for path analysis, and (2) uses abstract interpretation (AI) for the processor behavior analysis. In (1) the program becomes an ILP problem with structural (i.e., automatically extracted) and functional constraints (i.e., via analyses or manual annotations). In (2) the program blocks are classified w.r.t. their cache behavior two main types of behavior parameters - may/must hit/miss.

The ILP representation captures the flow information of a program, as ILP constraints. Therefore, the core component of this abstraction is the program counter, pc, wrapped in concr. The configuration of the ILP constraints extraction is in Fig. 4.

$$Cfg_{ILP} \equiv \langle \langle \langle Addr \rangle_{\mathsf{gaddr}} \langle Val \rangle_{\mathsf{ctridx}} \langle PC \mapsto K \rangle_{\mathsf{sconstr}} \rangle_{\mathsf{ilp}} \rangle_{\mathsf{abst}} \langle \langle Val \rangle_{\mathsf{pc}} \langle Addr \mapsto Val \rangle_{\mathsf{cmem}} \rangle_{\mathsf{concr}}$$
$$Cfg_{CA} \equiv \langle \langle K \rangle_{\mathsf{atype}} \langle PC \mapsto K \rangle_{\mathsf{collect}} \langle K \rangle_{\mathsf{jres}} \rangle_{\mathsf{abst}}$$
$$\langle \langle \langle Addr \mapsto Val \rangle_{\mathsf{ic}} \langle Addr \mapsto Val \rangle_{\mathsf{ages}} \rangle_{\mathsf{aic}} \langle Addr \mapsto Val \rangle_{\mathsf{prm}} \langle Addr \rangle_{\mathsf{faddr}} \langle Val \rangle_{\mathsf{ilen}} \rangle_{\mathsf{concr}}$$

**Fig. 4.** $Cfg_{ILP}$ - the $\mathbb{K}$ configuration for the ILP structural constraints extraction, and $Cfg_{CA}$ - the $\mathbb{K}$ configuration for the AI-based cache analysis.

The abstraction-specific part of $Cfg_{ILP}$ uses the sconstr cell to collect the ILP constraints and, for each program point, keeps the constraint index in the ctridx cell and the target address (for a branch/jump instruction), in the gaddr cell.

*Example 8.* In Fig. 5 we present the $\mathbb{K}$ rule to extract, for a branch instruction, the ILP constraints. The branch instruction *Ins* is at a program point *PC* and has the fall-through address *N* and the target address *J*. The flow information which

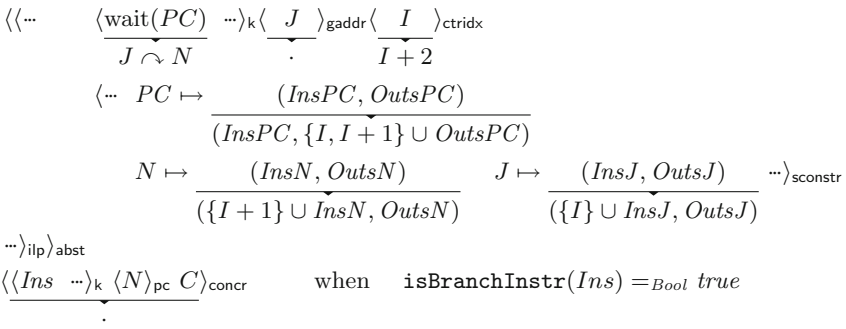


**Fig. 5.** Rule for structural ILP constraints generation from a branch instruction.

corresponds to the test instruction could be expressed as: the execution count of the condition is equal with the sum of execution counts of the two branches. This rewrite rule represents the case STOP in Fig. 2. If the current instruction is a test - as reflected by the condition "when `isBranchInstr`$(Ins)$"- then the computation in concr is stopped (i.e., it rewrites to "·") and the "processing" computation in abst starts. In the cell k of abst, the token "wait$(PC)$" identifies the program point where the concrete execution stopped. The token is transformed into a sequence of abstract computational tasks, for the instructions at addresses $J$ and $N$. The gaddr cell is emptied, because the corresponding input constraint for the jump address $J$ is generated at this step. This rewrite rule generates two new constraint indexes, $I$ and $I + 1$, hence the next available index is $I + 2$ (deposited in the cell ctridx). The constraints are collected in sconstr, where each program point is represented by its input $InsPC/N/J$ and output $OutsPC/N/J$ sets. The exit flow for the test at $PC$ is reflected in an updated $OutPC$ (with the two new indexes), while the entry flow for the two branches are in $InsN/J$. When the algorithm terminates, each program point has a set of constraints with the property that the sum of input constraints is equal to the sum of output constraints. □

The AI-based analysis for the instruction cache behavior computes sets of abstract cache states to classify each program instruction w.r.t. its cache activity. Hence, an instruction could be always-hit (after a must analysis) or always-miss (after a may analysis). The core of this abstraction is the pair of cache content, in ic and its corresponding age information, in ages. The associated join operations are based on intersection (for must analysis) and reunion (for may analysis) of abstract states. These abstractions are presented in details in [25]. In our methodology, the configuration $Cfg_{CA}$ for the instruction cache analysis is given in Fig. 4. The set of abstraction-specific cells include the analysis type (may or must) in atype, the result of the corresponding join operation, in jres, and the abstraction results in collect. Also, the two data abstractions and the cache behavior abstraction could include, in their abstraction-specific part of the configuration, a special cell cfg for a previously extracted control flow graph.

| **name** | **#lines** | ILP extraction | AI may instr. cache | AI must instr. cache |
|---|---|---|---|---|
| *icrc1* | 42 | 70.5%(647) | 23%(22422) | 22.1%(23287) |
| *duffcopy* | 104 | 70.8%(1519) | 23.4%(44064) | 19.5%(53368) |
| *expint* | 185 | 70.5%(2761) | 23.4%(91158) | 17.96%(120064) |
| *adpcm decode* | 312 | 71.08%(4403) | 23.81%(109560) | 12.91%(201942) |
| *adpcm encode* | 327 | 71.08%(4645) | 23.79%(119052) | 12.83%(220679) |

**Fig. 6.** Results for some Mälardalen programs for ILP structural constraints extraction, AI may-analysis for instruction cache, and must-analysis for instruction cache.

## 5 Implementation

We implement in $\mathbb{K}$-Maude [10] our general framework for resource-related analysis in the context of the WCET analysis. For illustration purposes, we opt to experiment with the ILP + AI method rather than the data abstractions, because of the more complex configurations, i.e., $Config_{ILP}$ and $Config_{CA}$.

The ILP monitors the execution between the language semantics and the main memory; a single token (an instruction fetch request) is necessary to stop the concrete execution. The AI part involves the cache memory module and, thus, it is more complex due to the parametrization based on the cache structure and functionality (i.e., cache size, associativity, or replacement policy) and due to the parametrization on the analysis type (i.e., may and must). Regarding the configuration size, the concrete system has 22 $\mathbb{K}$-cells, the ILP constraints extraction needs 9 $\mathbb{K}$-cells (with 4 abstraction-specific ones) and the cache analyses needs 21 $\mathbb{K}$ cells (with 12 abstraction-specific cells).

Moreover, we measure a reusability degree of the concrete specification, when various types of abstractions are integrated. The computation, after an abstraction integration, presents three kinds of rules: two kinds are coming from the concrete and the abstract specification and a third kind, generated by $\mathbb{K}$-Maude. When we measure the percentage of the abstract execution steps with respect to the total number of execution steps, we ignore the tool generated ones.

We select several of the Mälardalen benchmarks [13] and run on a 2.4 GHz Intel Core i5 MacBook Pro. We conduct the experiments on the following programs: the cycle redundancy check computation *icrc1*, the Duff device *duffcopy*, the exponential integral function computation *expint*, and the adaptive pulse code modulation algorithm of *encode* and *decode*. The size of each program is listed in the column **#lines** in Fig. 6.

The table in Fig. 6 shows the integration of the ILP constraints generation and the results on the AI-based instruction cache analyses (i.e., may and must). For each analysis, we present in its designated column the percentage of concrete rewrite rules in the execution and, in the bracket, the number of rewrites of this particular execution.

We observe a higher reusability percentage in the case of the ILP analysis than for the AI for cache analyses. The reason is that the ILP wrapped concrete configuration, $Cfg_{ILP}$, is simpler than the cache analyses configuration, $Cfg_{CA}$

(we recall that only the pc register and the code memory are necessary to detect the program flow). Running both may and must analyses under an implicit CFG produces results as low as 12 % reusability, for the *adpcm encode* benchmark program (on must analysis) and not higher than 24 % for the *adpcm decode* program (on may analysis). These results emphasize that, in the worst case scenario, when the join function is executed at every program point, there is still some percentage of the concrete specification that can be reused. Actually, this scenario proposes a lower bound for the reusability factor w.r.t. the system specification. However, running these analyses with an explicit CFG reduces drastically the number of join points and the reusability factor is as high as 85 % for the *adpcm encode* and *adpcm decode* programs.

## 6 Conclusions

In this paper we proposed a framework for modular resource analysis which relied on a trusted core formed by a modular formal executable semantics of the analyzed system. The work was motivated by the time resource and was constructed around analyses applied in WCET computation. The modularity aspect is both functional and structural. Namely, the functional modularity is at the level of the analysis and stemmed from a systematic integration of abstractions over the trusted core. In its turn, the structural modularity resides in the formal specification of the assembly language and the underlying architecture. We showed how to integrate into the resource analysis workflow the following WCET analyses: the constant propagation, the interval analysis, and the combined ILP+AI approach. We described the implementation and its testing w.r.t. an integration factor which we set to the reusability degree of the concrete specification in the abstract one. Our future work includes (1) a specification of pipeline behavior and analysis (requiring parameterized $\mathbb{K}$ modules - currently in development as described in [16]) and (2) further investigations on integrating other resource analyses into our proposed framework.

**Acknowledgments.** We thank very much to the anonymous reviewers for their comments and suggestions which helped us to focus and improve our work in this paper.

## References

1. AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers
2. Asăvoae, M., Asăvoae, I.M., Lucanu, D.: On abstractions for timing analysis in the $\mathbb{K}$ framework. In: Peña, R., van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2011. LNCS, vol. 7177, pp. 90–107. Springer, Heidelberg (2012)
3. Asavoae, M., Lucanu, D., Roşu, G.: Towards semantics-based WCET analysis. In: WCET (2011) (to appear)
4. Biere, A., Knoop, J., Kovács, L., Zwirchmayr, J.: The auspicious couple: Symbolic execution and WCET analysis. In: WCET, pp. 53–63 (2013)

5. Blazy, S., Maroneze, A., Pichardie, D.: Formal verification of loop bound estimation for WCET analysis. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 281–303. Springer, Heidelberg (2014)
6. Burger, D., Austin, T.M.: The SimpleScalar tool set, version 2.0. SIGARCH Comput. Archit. News **25**, 13–25 (1997)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM Press (1977)
10. Şerbănuţă, T.F., Roşu, G.: K-Maude: a rewriting based tool for semantics of programming languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 104–122. Springer, Heidelberg (2010)
11. Cutumisu, M., Onuczko, C., Szafron, D., Schaeffer, J., McNaughton, M., Roy, T., Siegel, J., Carbonaro, M.: Evaluating pattern catalogs: the computer games experience. In: ICSE, pp. 132–141 (2006)
12. Dams, D.: Comparing abstraction refinement algorithms. ENTCS **89**(3), 405–416 (2003)
13. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks: Past, present and future. In: WCET, pp. 136–146 (2010)
14. Hammond, K., Ferdinand, C., Heckmann, R., Dyckhoff, R., Hofmann, M., Jost, S., Loidl, H.W., Michaelson, G., Pointon, R.F., Scaife, N., Sérot, J., Wallace, A.: Towards formally verifiable WCET analysis for a functional programming language. In: WCET (2006)
15. Healy, C.A., Whalley, D.B., Harmon, M.G.: Integrating the timing analysis of pipelining and instruction caching. In: RTSS, pp. 288–297 (1995)
16. Hills, M., Roşu, G.: Towards a module system for K. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 187–205. Springer, Heidelberg (2009)
17. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7), 107–115 (2009)
18. Li, X., Mitra, T., Roychoudhury, A.: Accurate timing analysis by modeling caches, speculation and their interaction. In: DAC, pp. 466–471 (2003)
19. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: DAC, pp. 456–461 (1995)
20. Li, Y.T.S., Malik, S., Wolfe, A.: Efficient microarchitecture modeling and path analysis for real-time software. In: IEEE RTSS, pp. 298–307 (1995)
21. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7–8), 721–781 (2012)
22. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
23. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Syst. **37**(2), 99–122 (2007)
24. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010)
25. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise WCET prediction by separated cache and path analyses. Real-Time Syst. **18**(2/3), 157–179 (2000)
26. Wilhelm, R.: Why AI + ILP is good for WCET, but MC is Not, Nor ILP alone. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 309–322. Springer, Heidelberg (2004)

27. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem–overview of methods and survey of tools. TECS **7**(3), 1–53 (2008)
28. Wilhelm, R., Wachter, B.: Abstract interpretation with applications to timing validation. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 22–36. Springer, Heidelberg (2008)

Springer

http://www.springer.com/978-3-319-12465-0

Foundational and Practical Aspects of Resource Analysis
Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers
Dal Lago, U.; Peña, R. (Eds.)

2014, IX, 161 p. 34 illus., Softcover
ISBN: 978-3-319-12465-0