On the Modular Integration of Abstract Semantics for WCET Analysis

Mihail Asăvoae and Irina Măriuca Asăvoae

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

Abstract. In this paper we propose a modular WCET analysis which is built around a rewrite-based formal specification of an embedded system. Designing and analyzing embedded systems means that both hardware and software behavioral aspects should be considered. Such aspects are captured using the notion of system configuration. The modular WCET analysis uses a configuration-based design methodology, which is then instantiated to accommodate data and flow abstractions. The instantiations require no modifications of the original rewrite-based formal specification. We implement the configuration-based modular WCET analysis, and investigate its behavior with respect to a reusability metric.

1 Introduction

Interaction between an embedded real-time system and the external environment could be stated as a set of timing constraints. To ensure the system correctness w.r.t. timing behavior, a schedulability analysis, which requires worst case execution times (WCETs) of the component tasks, is performed over this set of constraints. The quality of the schedulability depends on the quality of the computed WCETs, therefore, apart from their intrinsic guarantee w.r.t. safety, they also should be tight.

A method for timing (WCET) analysis attempts to determine execution (upper) timing bounds of a program running on a specified architecture [21]. Existing approaches exploits this natural modularity, providing solutions for the path analysis and respectively, for the processor behavior analysis. A path analysis problem concerns with finding the longest execution path regardless of the architecture behavior, while a processor behavior analysis computes invariants wrt the instructions' behavior in the presence of the underlying architecture. A successful approach for WCET analysis proposes an integer linear programming (ILP) solution [18, 15] for the path analysis and uses abstract interpretation [7] for various micro-architecture elements or combinations of them [12, 14, 19, 20]. ILP + AI method achieved industrial success through tools such as `aiT` tool [1].

We approach the timing analysis problem from a slightly different perspective. We rely, as introduced in [3], on a formal definition of the Simplescalar PISA assembly language [4] as the central piece of a framework for timing analysis. A language formal definition (syntax + semantics) contains all the necessary information that could be used to build specialized analyses over it. Therefore, the well-tested formal executable semantics should serve as a trusted kernel for a timing analyzer. We also define parametric instruction and data caches behaviors to serve as starting point for processor behavior abstractions. Our design goal is to keep this kernel unmodified when we define the program and architecture analyses. We use a specialized, rewrite-based framework, called $\mathbb{K}$ [17], which allows explicit representation and manipulation of program configurations. $\mathbb{K}$ comes with a concise notation to define the rewrite rules, using a novel concept called context transformation [13]. We elaborate more on this in the next section.

In [2] we present a rewrite-based encoding of the ILP+AI approach for timing analysis. In this paper we show how this method, as well as constant propagation and interval analysis, are instances of a general methodology of abstractization via configuration. We start with the specification of the entire system, arranged into separate modules which communicates among each others via messages (functions). Since we work in a rewriting environment, each module is a rewrite-logic theory, consisting, in our case of a set of equations and rewrite rules. Also, each module contains a sub-configuration that, together with the ones from the imported modules, has the necessary information to define the behavior. We name this the concrete representation of the system. Further, we propose a meta-algorithm to generate abstract rule schemata for timing analysis abstractions. This algorithm starts with an abstract configuration that contains a wrapped concrete sub-configuration and the messages of interest that are to be monitored during the abstract execution. The general idea is that the abstraction monitors the flow of execution as it is defined by the formal executable semantics and the architecture description. When a message of interest occurs, the abstract execution stops the concrete counterpart and process the available information. There are three kinds of rewrite rules that are generated: one for the initialization of the abstract execution, a set of rules to stop the concrete execution and another set to resume it.

Our modeling methodology is based on this context transformation and the inherited modularity of the $\mathbb{K}$ framework. The generic modular system, where the constitute modules communicate through a message passing mechanism, uses the underlying $\mathbb{K}$ framework technology - through a special $\mathbb{K}$ cell called k which plays a data bus role. First, we discuss the design aspects of our approach and then the analysis part.

Design - We take the general configuration C, which comprises of all semantic entities of the specified system and split it into the sub-configurations $C_1, C_2, \ldots,$ C_n, each C_i, included into a separate $\mathbb{K}$ module. The k cell is included into each of the C_i sub-configurations and it is used for the computational purpose and to pass messages among modules and to facilitate, in this way, the interaction between modules. These messages play the role of meta-assertions between modules. Therefore, a concrete execution in this system is an interleaving between computations and messages that are interchanged between modules.

Analysis - We define abstractions in the following way: wrap the concrete sub-configurations of interest into a configuration $wrap(C_i)$, and define the abstract configuration A, which has as sub-configuration $wrap(C_i)$. The initial split of C into C_is allows, when the abstraction is applied, to execute only the system's functionality that is of interest. Moreover, the abstraction controls the "concrete" execution and enables functionality reuse. A different consequence of this modeling methodology is that it permits to implant certain characteristics of the concrete semantic entities.

Before we present an instance of our design methodology for the worst case execution time (WCET) analysis of embedded programs, we motivate our need for such a general modular design. A method for timing (WCET) analysis attempts to determine execution (upper) timing bounds of a program running on a specified architecture [21]. Existing approaches exploits this natural modularity, providing solutions for the path analysis and respectively, for the processor behavior analysis. A path analysis problem is concerned with finding the longest execution path regardless of the architecture behavior, while a processor behavior analysis computes invariants with respect to the instructions' behavior in the presence of the underlying architecture.

While our methodology is general, we perform timing analysis of assembly programs executed in the presence of instruction and data caches and a main memory system. Our initial configuration C consists of the semantic entities to define the assembly language as well as the micro-architecture components. A natural split produces sub-configurations (and modules) for the language, the instruction cache, the data cache and the main memory system. Now, the language module plays the role of the processor, as it requests information from the memories. By wrapping language-specific semantic entities, the instruction cache and the main memory models, we encode well-known abstract interpretation [7] based analyses such as constant propagation [6], interval analysis [16] as well as may and must analyses for caches [19], for the purpose of timing analysis [21]. We implement these cache behavior abstractions as part of one of the successful WCET analysis method - the ILP+AI approach.

The modular timing analysis is prototyped in $\mathbb{K}$-Maude [8], the $\mathbb{K}$ framework implementation on top of Maude system [5]. The experimental results are presented with respect to a reusability metric. We measure the implementation from the following perspective: keeping the formal semantics, how much of the concrete system is used, while the abstractions are applied. We select and conduct experiments on a subset of the Mälardalen benchmarks [9].

Related Work In [11] it is presented a specification, in rewriting logic, of a simple pipelined microprocessor. The goal is to formally verify its behavior. The underlying architecture is called SPM, with only five instructions (one for each class of instructions) and separate code and data memory modules, to accommodate further extensions to instruction and data caches. This modeling also includes a register file and the special register for the program counter. This instance for timing analysis addressed a similar architecture with several notable differences. We start with the similarities.

The two approaches consider similar semantic entities such as code and data memory, the register file and the program counter. All these form the core of an instruction set architecture specification. With respect to the underlying module representation, the work in [11] use Maude, while our $\mathbb{K}$ definition is also compiled into Maude modules. Another similarity is that there are application specific modules: in the SPM case it is about a module that encode time consistency and one-step theorems, used to prove the correctness of the microprocessor, while, in our case, there are the abstraction related modules, as well as a module with semantic operations (i.e. builtin). A final, similar point is that both approaches are parametric. SPM is parametric in the number of registers, the memory address and the word size, while our system is design to be parametric in the underlying builtin support, to facilitate abstraction definitions. Next, we discuss the differences. First, the assembly language of SPM is reduced to a minimum, while we propose, in the language semantics module a complete implementation of a RISC language of 120 instructions. Second, the definition of SPM considers a monolithic representation of processor and main memory related concepts. We achieve a better modularity when we split the register file and program counter, in the language semantics module from the main memory.

With respect to the timing analysis, a successful approach for WCET analysis proposes an integer linear programming (ILP) solution [18, 15] for the path analysis and uses abstract interpretation [7] for various micro-architecture elements or combinations of them [12, 14, 19, 20]. ILP+AI method achieved industrial success through the `aiT` tool [1]. In [2] we present a rewrite-based encoding of the ILP+AI approach for timing analysis. In this paper we describe this encoding as an instance of the methodology, and present several experimentation results, with respect to the quality of the encoding.

Paper outline This paper is organized as follows. Section 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, from the configurations' perspective and respectively present several key details used to design our semantics-based instance for timing analysis. Section 5 covers some implementation and experimental details with the current prototype while Section 6 contains the conclusions.

2 Preliminaries

2.1 The $\mathbb{K}$ framework

The $\mathbb{K}$ framework is a rewrite-based framework specialized in the design and analysis of programming languages. A definition in the $\mathbb{K}$ framework consists of *configurations*, *computations*, and *rules*. The configurations, formed of $\mathbb{K}$ cells, are labeled and nested structures that represent program states. A particular cell, called k holds the list of computational tasks and it is responsible for the computation. An example of a simple configuration for an assembly programming language could contain the register file regs, a special register for program counter pc and a representation of the memory mem. Using the $\mathbb{K}$ notation, we write it as follows:

$$\langle K\rangle_{\mathsf{k}}\langle \mathit{Reg}\rangle_{\mathsf{pc}}\langle \mathit{Map}[\mathit{Reg} \mapsto \mathit{Val}]\rangle_{\mathsf{regs}}\langle \mathit{Map}[\mathit{Addr} \mapsto \mathit{Val}]\rangle_{\mathsf{mem}}$$

where *Reg*, *Val* and *Addr* are sorts for registers, stored values and respectively memory addresses.

The rules in $\mathbb{K}$ are divided into two classes: *computational rules* that may be interpreted as transitions in a program execution, and *structural rules* that modify a term to enable the application of a computational rule. One example of a computation rule is the operation of a register look-up, which processes this request in the k cell, by accessing the regs cell for the value.

$$\langle\ R => V\ \ \ldots\rangle_{\mathsf{k}}\ \langle\ldots\ R \mapsto V\ \ \ldots\rangle_{\mathsf{regs}}\ \ [\mathtt{LU-rule}]$$

Such $\mathbb{K}$ rewrite rule uses an ingenious notation to identify where the rewriting is taking place, at the cell level. For example, in case of the `LU-rule`, the top of the k cell is rewritten, everything else remains the same. The cell regs holds a mapping from registers to values. The ellipses show that the element with the key R is somewhere in the map, not necessarily the first nor the last. It is important to notice that the other cells in the semantics, pc and mem are not represented explicitly in the rewrite rule. $\mathbb{K}$ relies on a rule completion mechanism, called configuration abstraction, that adds the remaining part of the configuration, in this case the cells pc and mem.

We use this form of abstraction for two purposes: first, to split the global configuration into smaller sub-configurations (corresponding to modules) and second, to allow cell wrapping. The latter facilitates, via matching, to abstract away the unnecessary semantic entities. We present these in Section 5, where we describe the implementation.

2.2 Abstract Interpretation

Abstract interpretation was introduced in [7] and since then, it 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 several abstract semantics, connected via the abstraction relation. Abstract interpretation has at its core the notion of collecting semantics of a program. A collecting semantics assigns to each program point the set of states that may occur during any execution. While, in general the collecting semantics is not computable or it cannot be efficiently computed, in verification a la abstract interpretation, the collecting semantics is safely approximated. The abstractions in abstract interpretation are generic, in the sense that they are applied to the entire class of programs, for a programming language. Next we introduce the elements of the abstract interpretation following the approach in [22].

The base of a program analysis a la abstract interpretation is the collecting semantics, a rich semantics, from which all the other abstract semantics are derived.

A program analysis relies on the following two elements: an abstract domain and an abstract semantics. The abstract domain is defined by a complete semi-lattice, 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 function is defined in terms of these representation functions and together with the concretization function form a Galois connection, if certain conditions are satisfied by this pair of functions. An important part is the abstract semantics which re-implements the transfer functions of the concrete semantics; of course there are several requirements for these functions. The abstract semantics is used to solve the program analysis problem, using a fixpoint style computation.

3 A General System Design

Let us consider a complex computational system, specified in $\mathbb{K}$ that has the general configuration, denoted as C_{global}, which comprises of all semantic entities that are necessary to capture the system behavior. Following a top-down approach, we take the C_{global} configuration and split it into a number of sub-configurations: $C_1, C_2, \dots, C_n$, not necessarily disjoint. There are several important observations with respect to this form of abstraction. First, at an informal level, each sub-configuration should be used to capture a well-defined component of the system. Second, at an organizational level, the configuration splitting determines two kinds of modules:

– *structural module*: the special computational cell k is not included in the module's sub-configuration
– *functional module*: the special computational cell k is included in the module's sub-configuration, being actively involved in the overall system behavior

Each sub-configuration C_i is included into a separate module, and we return to the implications of this aspect latter in this section. The k cell that is included into C_i sub-configurations are used, apart from the computational purpose, to pass messages among modules and to facilitate, in this way, the interaction between modules. Therefore, a concrete execution in this system is an interleaving between computations and messages that are interchanged between modules. We work under the following assumption: the nature of the computation in our setting is exclusively sequential - any message that is sent from a module is received by exactly one other module. Of course, this model is amenable to extensions, for example, to accommodate concurrent exchanges of messages.

Before we present in detail the components of this modular system for applications running in embedded environment, we refer to our main design target - to

```
Input:
    $M_i$ with $i = 1..n$ - a collection of $\mathbb{K}$ modules
                         with corresponding sub-configurations $c_i$
    $C_{j,k}$ with $j, k = 1..n,\ j \neq k$ - the set of messages between modules
                                   $M_i$ and $M_j$
    $\langle \langle K \rangle_{\mathsf{k}}\ Rest \rangle_{\mathsf{abst}}$ - the abstract configuration, with $Rest$
                                   being abstraction specific
    $\langle\ \rangle_{\mathsf{concr}}$ - the wrapped concrete configuration

Output:
    $S$ - the abstract skeleton (a set of partially defined $\mathbb{K}$ rewrite rules

INIT
  $S \leftarrow S \cup \{\langle\langle init => busy\rangle_{\mathsf{k}} Rest\rangle_{\mathsf{abst}}\ (. => \langle\langle C_{1,i}(init)\rangle_{\mathsf{k}} Rest\rangle_{\mathsf{concr}})\}$
STOP
forall $C_{i,j}$ generate a rule skeleton as follows:
  $S \leftarrow S \cup \{\langle\ \_\ \rangle_{\mathsf{abst}}\ (\langle\ \langle\ C_{i,j}\ =>\ .\ \rangle_{\mathsf{k}}\ c_i\ \rangle_{\mathsf{concr}})\}$
RESUME
forall $C_{i,j}$ generate a rule skeleton as follows:
  $S \leftarrow S \cup \{\langle\ \langle busy\ =>\ .\ \rangle_{\mathsf{k}}\ \rangle_{\mathsf{abst}}\ (\langle\ \langle\ C_{i,j}\ \rangle_{\mathsf{k}}\ c_i\ \rangle_{\mathsf{concr}})\ =>\ .\}$
```

Fig. 1. Methodology to generate a rule schemata

facilitate the process of defining abstractions. With respect to this aspect, we define abstractions in the following way: first wrap the concrete sub-configurations of interest into a configuration $wrap(C_i)$, and then define the abstract configuration A, which has as sub-configuration $wrap(C_i)$.

It is important to notice that the $\mathbb{K}$ framework machinery (or the configuration abstraction) complete the partially defined specification. In this way, we make a seamless integration of the concrete specification in an abstract execution environment. Going with the intuition even further, this form of abstraction poses a side advantage. Because of the wrapped $\mathbb{K}$ cells, during the execution of the $\mathbb{K}$ specification, only certain $\mathbb{K}$ rules are matched. Moreover, the abstraction controls the "concrete" execution and enables functionality reuse. A different consequence of this modeling methodology is that it permits to implant certain characteristics of the concrete semantic entities (i.e. timing or power consumption)

We present this methodology as a meta-algorithm, in Fig. 1, then we show how we create an instance for the WCET analysis of embedded programs. The part of abstraction encoding considers only a set of modules M_i of the general system, each module having its representative sub-configuration wrapped into the abstract configuration. These modules communicate via a set of messages $C_{j,k}$ that are sent through the computational cell k. Note that the set of structural modules are not explicitly represented in the set M_i, but they are freely used, if their corresponding sub-configuration is part of the system's of interest configuration. The messages $C_{j,k}$ become points of interest in the abstract execution. Basically, the abstraction monitors the concrete execution and, when an event of interest is placed in the k cell, it stops the execution and process (i.e. collects, updates) the available information. The algorithm produces a rule schemata as a set S of $\mathbb{K}$ rewrite-rules, for all the involved modules and messages.

This meta-algorithm produces the rule schemata in three stages, named INIT, STOP and RESUME, after the actions of interest on the concrete execution of the system. All the rules comply to the $\mathbb{K}$ notation and have two parts: abstract, represented by the abst cell and concrete, represented by the concr cell. Since the meta-algorithm generates a rule schemata on how to interleave these two executions, the rules take into consideration the k cell, the only that is responsible with the computation.

During the INIT stage, the abstract execution is initialized in two steps. The abst cell has on top of its computation a special token called init that symbolically represents the initial content of this cell. init is rewritten into another special token, called busy, which signals the execution's transfer to the concrete counterpart, in the concr cell, which is just created. The content of the corresponding k cell in concr has the message $C_{1,i}$ from the start module to all the modules, i, that accept this message. The execution of an instance of the meta-algorithm (an example is presented in the next section) is an interleaving between computations taking place in the k cell of the abst and concr cells. The second stage STOP presents the set of rules corresponding to stopping the execution in the wrapped concrete part of the configuration. Regardless of the information in abstr, the message $C_{i,j}$ on top of the computation in concr is consumed. The third part of the meta-algorithm, RESUME, show the set of rules corresponding to the dual action of stopping the execution, with the concr cell being dissolved. The execution continues in the abstract cell with what remains of the computation and eliminating the special token busy. It is important to notice that, for an abstract interpretation based analysis encoded as an instance of this meta-algorithm, we do not select a particular abstract state to carry it back into the concrete part and execute it there.

4 Modular Timing Analysis

4.1 The System

The $\mathbb{K}$ framework enables the modular and executable semantics specification of programming languages. The key ingredient to system specification using $\mathbb{K}$ is through configuration manipulation. Our proposed system considers, along with the formal executable semantics of the assembly language, parametric specifications for behaviors of instruction and data cache memories, as well as for a main memory system. The parametrization refers to both the cache structure (i.e. size, associativity) and functionality (i.e. replacement policies, writing policies).

The overall system consists of a set of $\mathbb{K}$ modules, shown in Fig. 2. We discuss next the functional modules Language Semantics Module, IC Module and Main Memory, used in the abstraction encodings, as in Section 4.2 and Section

4.3. For each of the enumerated modules we present the corresponding concrete configurations, which from the methodology perspective are wrapped into the concr cell (in Fig. 1).

Language Semantics - The complete $\mathbb{K}$ configuration for the assembly language is:

$$Config_{Lang} \equiv \langle K \rangle_{\mathsf{k}}\, \langle Reg \rangle_{\mathsf{pc}}\, \langle Reg \rangle_{\mathsf{lo}}\, \langle Reg \rangle_{\mathsf{hi}}$$

$$\langle Reg \rangle_{\mathsf{ra}}\, \langle Bool \rangle_{\mathsf{fcc}}\, \langle Bool \rangle_{\mathsf{break}}\, \langle \mathsf{Map}[Reg \mapsto Val] \rangle_{\mathsf{regs}}\, \langle \mathsf{Map}[FReg \mapsto Val] \rangle_{\mathsf{fregs}}$$

The language configuration $Config_{Lang}$ has the computational cell k, the program registers and two flags - break and fcc for abrupt termination of execution and respectively a floating point parameter. The necessary language registers are grouped into general-purpose registers (i.e. integer in regs and floating point in fregs) and special registers. These latter ones include the program counter value - the pc cell, the multiplication/division result registers - the lo and hi cells, the return address of a function call - the ra cell. Each $\mathbb{K}$ cell displays its corresponding 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.

$$Config_{MM} \equiv \langle K \rangle_{\mathsf{k}}\, \langle \mathsf{Map}[Addr \mapsto Val] \rangle_{\mathsf{cmem}}\, \langle \mathsf{Map}[Addr \mapsto Val] \rangle_{\mathsf{dmem}}$$

The concrete main memory configuration $Config_{MM}$ plays an important role in both control and data abstractions because it stores the input program.

IC Memory - The configuration for the instruction cache captures both organizational and structural semantic aspects:

$$Config_{IC} \equiv \langle K \rangle_{\mathsf{k}}\, \langle \mathsf{Map}[Addr \mapsto Instr] \rangle_{\mathsf{ic}}\, \langle \mathsf{Map}[Parm \mapsto Val] \rangle_{\mathsf{param}}\, \langle Addr \rangle_{\mathsf{repl}}$$

$$\langle \mathsf{Map}[Addr \mapsto Val] \rangle_{\mathsf{ages}}\, \langle \mathsf{Map}[Type \mapsto Val] \rangle_{\mathsf{profile}}\, \langle Val \rangle_{\mathsf{ilen}}\, \langle Addr \rangle_{\mathsf{faddr}}$$

The ic and ages cells contain the instruction cache blocks and respectively their associated age information. The repl cell has the address of the cache block as the next eviction candidate (computed based on the blocks age). The profile and param cells store the hit/miss counts and respectively the cache parameters (i.e. size, associativity), while the ilen and faddr are the instruction length and respectively the address of the first instruction in the program.

This system is designed to simulate the execution of a program on an underlying architecture, with the $\mathbb{K}$ module corresponding to the language semantics plays the role of the processor. We call this the *concrete execution* of the system. The execution of one instruction consists of a sequence of code and data requests to the memory system, which in turn is represented by one or several $\mathbb{K}$ modules (IC Module, DC Module and Main Memory in Fig. 2).

The modularity of our design allows us to define abstractions directly over the existing modules, which 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

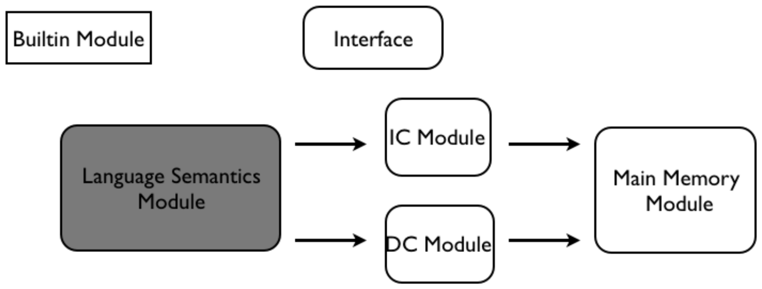

Fig. 2. WCET Analyzer - The Module System

next on how we encode four abstractions: constant propagation and the interval analysis in Section 4.2 and the ILP+AI combined method in Section 4.3. We present these encodings from the abstract configuration point of view - what concrete semantic entities are retained (in a concr cell) and what abstraction specific entities are required (in an abst cell).

4.2 The Data Abstractions

The constant propagation and the interval analysis are widely used static analyses for WCET analysis, according to [22]. In this section we discuss how the executions of the constant propagation and interval analysis are integrated within our methodology.

A constant propagation analysis produces, at each program point in the program, the set of variables (i.e. register values) which have constant values. According to the general scheme for an abstract interpretation based analysis, presented in Section 2.2, we need to define the abstract domain and abstract versions of the language operations. The unknown value of a variable should be represented in the abstract domain by a special variable. Also, with respect to the abstract versions of the operation, these are extensions of the concrete ones, extended with the unknown value case.

This analysis requires, from the concrete system, the language registers and the code memory, thus these semantic entities are wrapped in the abst cell. The abstraction-specific component is empty. Therefore, the configuration for the constant propagation, restricted to the integer domain, is the following:

$$Config_{CP} \equiv \langle\ \langle \mathsf{Map}[Reg \mapsto Val] \rangle_{\mathsf{regs}}\, \langle Reg \rangle_{\mathsf{lo}}\, \langle Reg \rangle_{\mathsf{hi}}\, \langle Val \rangle_{\mathsf{break}}$$

$$\langle \mathsf{Map}[Addr \mapsto Val] \rangle_{\mathsf{cmem}}\ \rangle_{\mathsf{abst}}$$

The interval analysis [16] relies on an abstract representation of a language value as an interval. Our assembly language uses several types of values, integers and floating points, and the interval-based values applies to all of them. Next, we refer to the interval analysis encoding on the integer domain. The interval analysis generalizes the constant propagation and we follow the same design steps as previously, from [22]. We start with the abstract domain, and in the case of the interval analysis, we need to define new sorting information to capture the interval bounds. This typing integration is seamless with respect to the semantics rules in language semantics module, but the underlying builtin module should be modified.

Once the abstract domain and the new builtin module are defined, we proceed to integrate the interval analysis in the same spirit. Since this analysis generalizes the constant propagation, there is the following relations between Config_{CP} and Config_{CP} (defined below). The abst cell content of Config_{CP} remains the same (modulo interval-related sorting) and there is an abstraction-specific cell, called mdop. The configuration for the interval analysis is defined as follows:

$$\mathit{Config}_{IA} \equiv \langle\ \langle \mathsf{Map}[\mathit{Reg} \mapsto \mathit{Val}] \rangle_{\mathsf{regs}} \langle \mathit{Reg} \rangle_{\mathsf{lo}} \langle \mathit{Reg} \rangle_{\mathsf{hi}} \langle Val \rangle_{\mathsf{break}}$$

$$\langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Val}] \rangle_{\mathsf{cmem}}\ \rangle_{\mathsf{abst}} \langle Val \rangle_{\mathsf{mdop}}$$

The encoding of the interval analysis requires this particular cell mdop because of the multiplication and division operations. The assembly language 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). A problem with the interval analysis is the rounding errors for lower and upper bounds of the interval result, after such arithmetic operation. In the case of the multiplication of two intervals, we use the mdop cell to store the result, as an intermediate stage, which is then split and load into registers hi and lo from abst.

4.3 The ILP+AI Approach

A successful approach for WCET analysis proposes (1) an integer linear programming (ILP) solution, for the path analysis and (2) uses abstract interpretation (AI) for various micro-architecture elements or combinations of them. Point (1) considers the program represented as an ILP problem, with two kinds of constraints: structural, which can be automatically extracted from the code and functional, which could be the result of manual annotations. Point (2) classifies the memory blocks used by the program, with respect to their instruction and data cache behavior.

This respective ILP representation captures the flow information of the input program, as integer linear constraints. Therefore, the core component of this abstraction is the program counter value, pc, wrapped in the abst cell. The configuration of the ILP structural constraints extraction is the following:

$$\mathit{Config}_{ILP} \equiv \langle\ \langle \mathit{Reg} \rangle_{\mathsf{lo}} \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Val}] \rangle_{\mathsf{cmem}}\ \rangle_{\mathsf{abst}}$$

$$\langle \langle \mathit{Addr} \rangle_{\mathsf{gaddr}} \langle \mathit{Val} \rangle_{\mathsf{ctridx}} \langle \mathit{Map}[PC \mapsto K] \rangle_{\mathsf{sconstr}} \rangle_{\mathsf{ilp}}$$

The abstraction-specific component of Config_{ILP} contains a $\mathbb{K}$ cell called sconstr to collect the ILP constraints, for each program point PC. The currently generated constraint index is in ctridx, and the target address, in case of a jump/branch instruction is in gaddr.

The AI-based analysis for the instruction cache behavior computes sets of abstract cache states to classify each program instruction with respect to its cache activity. For example, two classes of interest are: always-hit (result of a must analysis) and always-miss (result of 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).

The configuration for the instruction cache behavior analysis is the following:

$$\mathit{Config}_{CA} \equiv \langle\ \langle\ \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Val}]\ \rangle_{\mathsf{ic}}\ \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Val}]\ \rangle_{\mathsf{ages}}\ \rangle_{\mathsf{aic}}$$

$$\langle \mathit{Map}[K \mapsto \mathit{Val}] \rangle_{\mathsf{params}}\ \rangle_{\mathsf{abst}} \langle K \rangle_{\mathsf{atype}} \langle \mathit{Map}[PC \mapsto K] \rangle_{\mathsf{collect}} \langle K \rangle_{\mathsf{jres}} \langle \mathit{Addr} \rangle_{\mathsf{faddr}} \langle\ \mathit{Val} \rangle_{\mathsf{ilen}}$$

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.

A final observation, 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 computed control flow graph of a program.

The execution of a program over this wrapped configurations is called *the abstract execution*. In the next section, we present implementation specific details of the ILP+AI methodology for WCET analysis, starting with a core implementation of the system.

5 Implementation

In this section we discuss the implementation of an instance of the proposed methodology, for timing analysis, presented in Section 4.1, from the perspective of the ILP+AI method integration. For presentation purposes, we opt to present this particular method rather than the two data abstractions, because of the more complex configurations involved - Config_{ILP} and Config_{CA}. We use $\mathbb{K}$-Maude [8], a prototype implementation of the $\mathbb{K}$ framework done on top of Maude system [5].

The ILP part monitors the execution between the language semantics and the main memory module, having to stop the concrete execution on one single message - instruction fetch request, while the AI part is more complex because the system execution flow involves the cache memory module. Moreover, the AI instance is parameterized by the cache structural and functional aspects (i.e. cache size, associativity, replacement policy etc) and the corresponding join operations for the may and must analyses are implemented in a generic fashion.

Since we build the abstractions over the concrete part, we plan to measure the degree of reusability of the concrete specification during the abstract computation. During our abstract computation (either for the ILP or AI part), the

name	locations	concrete (%)	abstract (%)	total rewrites
icrc1	42	70.5	29.5	647
duffcopy	104	70.8	29.2	1519
expint	185	70.5	29.5	2761
adpcm decode	312	71.08	28.92	4403
adpcm encode	327	71.08	28.92	4645

Fig. 3. Results for the ILP structural constraints extraction

name	concrete (%)	abstract (%)	total rewrites
icrc1	23	77	22422
duffcopy	23.4	76.6	44064
expint	23.4	76.6	91158
adpcm decode	23.81	76.19	109560
adpcm encode	23.79	76.21	119052

Fig. 4. Results for the AI-based (may) analysis for instruction cache behavior

name	concrete (%)	abstract (%)	total rewrites
icrc1	22.1	77.9	23287
duffcopy	19.5	80.5	53368
expint	17.96	82.04	120064
adpcm decode	12.91	87.09	201942
adpcm encode	12.83	87.17	220679

Fig. 5. Results for the AI-based (must) analysis for instruction cache behavior with implicit CFG

execution trace contains three kinds of rewrite rules and equations: two types - those which belong to the concrete and respectively the abstract specification of the system and one type - those 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. Hence, the total number of rewrites, which appear under the column **total rewrites** in all three tables in Fig. 3, 4 and 5, consists only of the rewrite rules and equations of our specification.

We select several of the Mälardalen benchmark programs [10] 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 of the programs is listed under the column **locations** in Fig. 3.

The first table records the results on the abstract execution for the ILP structural constraints generation. The third column **concrete** and the forth column

abstract records the percentages of $\mathbb{K}$ rewrite rules and equations that belong to the concrete, respectively abstract part of our system. The high percentage of reusability of concrete code is because the wrapped concrete configuration is simpler $Config_{ILP}$ than the one for cache behavior analysis (we recall that only the pc register and the code memory are necessary to detect the program flow). Also, this analysis does not rely on external functions, such as join operations, on whose results we discuss next.

The tables in Fig. 4 and Fig. 5 display the experimental results on the abstract interpretation-based analyses, may and respectively must, for the instruction cache behavior. Running both may and must analyses, having an implicit representation of the CFG yields results in terms of reusability as the percentages get as low as 12% 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, we define this as the lower limit for the reusability factor, with respect to the specification (both concrete and abstract). Running the same analyses with an explicit CFG reduces drastically the number of join points and the reusability factor hits 85% for the *adpcm encode* and *adpcm decode* benchmark programs.

6 Conclusions

In this paper we proposed a modular WCET analysis which relies on a two-phased $\mathbb{K}$-based methodology for designing and reasoning about embedded systems. The first component allowed separation of concerns at the level of design, while the second component was concerned with encoding abstractions over the particular system. The concrete description of a system provided the base for abstraction integration in the WCET analysis workflow. We encode the constant propagation, the interval analysis and the combined ILP+AI approach, involving both program and processor behavior analyses. We tested the current implementation on several benchmark programs, with the formal semantics kept unmodified, measuring a factor of reusability of trusted, concrete definition.

References

1. AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers
2. Asavoae, M., Asavoae, I.M., Lucanu, D.: On abstractions for timing analysis in the k framework. In: FOPARA. LNCS, vol. 7177, pp. 90–107 (2011)
3. Asavoae, M., Lucanu, D., Roşu, G.: Towards semantics-based wcet analysis. In: WCET (2011), to appear
4. Burger, D., Austin, T.M.: The simplescalar tool set, version 2.0. SIGARCH Comput. Archit. News 25, 13–25 (June 1997)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to

Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350. Springer (2007)

6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 269–282. ACM Press, New York, NY, San Antonio, Texas (1979)
7. 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)
8. Şerbanuţă, T.F., Roşu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: WRLA 2010. LNCS, vol. 6381, pp. 104–122 (2010)
9. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The mälardalen wcet benchmarks: Past, present and future. In: WCET. pp. 136–146 (2010)
10. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The mälardalen wcet benchmarks: Past, present and future. In: WCET. pp. 136–146 (2010)
11. Harman, N.A.: Verifying a simple pipelined microprocessor using maude. In: Selected papers from the 15th International Workshop on Recent Trends in Algebraic Development Techniques. pp. 128–151. WADT '01, Springer-Verlag (2001)
12. Healy, C.A., Whalley, D.B., Harmon, M.G.: Integrating the timing analysis of pipelining and instruction caching. In: IEEE Real-Time Systems Symposium. pp. 288–297 (1995)
13. Hills, M., Rosu, G.: Towards a module system for k. In: WADT. pp. 187–205 (2008)
14. Li, X., Mitra, T., Roychoudhury, A.: Accurate timing analysis by modeling caches, speculation and their interaction. In: DAC. pp. 466–471 (2003)
15. 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)
16. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM (2009)
17. Roşu, G., Şerbănuţă, T.F.: An overview of the k semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
18. tsun Steven Li, Y., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: in Proceedings of the 32nd ACM/IEEE Design Automation Conference. pp. 456–461 (1995)
19. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise wcet prediction by separated cache and path analyses. Real-Time Systems 18(2/3), 157–179 (2000)
20. Wilhelm, R.: Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In: VMCAI. pp. 309–322 (2004)
21. 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. ACM Trans. Embed. Comput. Syst. 7(3), 1–53 (2008)
22. Wilhelm, R., Wachter, B.: Abstract interpretation with applications to timing validation. In: CAV. pp. 22–36 (2008)