SEMANTICS-BASED WCET ANALYSIS

MIHAIL ASAVOAE

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE
UNIVERSITY "ALEXANDRU IOAN CUZA" IASI

2012

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree of qualification at this or any other university or institution of learning.

Abstract

This dissertation presents the design of a definitional semantics-based WCET analyzer, bridging the gap between the principles of formal executable specification, promoted by the $\mathbb{K}$ framework and the existing methods and techniques, which were successfully applied in analysis and verification of embedded software. The standard view on WCET analysis considers the set of all possible executions of a given program on a specified, underlying hardware. Therefore, there is a projection of design and implementation methods at both the level of the program and the architecture. We present a novel, unified view on this projection, using the $\mathbb{K}$ framework definitional power (i.e. via configuration representation and manipulation). We define a formal executable semantics of a RISC assembly language and a parametric specification for micro-architecture behavior, namely for instruction and data caches. In this way, a program can be formally executed, in concrete, on a family of architectures (i.e. with respect to several design parameters). This combined system is modular, consists of a number of communicating modules corresponding to hardware and software components. Moreover, this organization allows us to investigate, at the level of individual or group of modules, how to embed, definitionally, various abstractions for timing analysis. We present embeddings for program-related abstractions such as extraction of a control-flow graph, generation of structural integer linear programming constraints, constant propagation and interval analysis. Also, we present the integration of hardware-related abstractions for cache behavior prediction. Our work fully adheres to the principles of design of programming languages and their afferent analysis tools, as advertised by $\mathbb{K}$ framework — formal definition of a programming language should be used unmodified in program analysis and verification.

List of Publications

Revised publications directly related with the dissertation

1. Mihail Asăvoae, and Dorel Lucanu and Grigore Roşu
   Towards Semantics-Based WCET Analysis
   Proceedings of the 11th International Workshop on Worst-Case Execution-Time Analysis (WCET2011)
   editor Christopher Healy
   Ed. Austrian Computer Society (OCG). (to appear)
   in Oasics Series, Schloss Dagstuhl (to appear)

2. Mihail Asăvoae, and Irina Măriuca Asăvoae:
   Using the Executable Semantics for CFG Extraction and Unfolding
   13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2011, Timisoara, Romania, 26-29 September 2011,
   editors Dongming Wang, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt and Daniela Zaharie,
   pages 123–127, IEEE Computer Society, 2011,
   ISBN 978–1–4673–0207–4.

3. Mihail Asăvoae, Irina Măriuca Asăvoae, and Dorel Lucanu:
   On Abstractions for Timing Analysis in the $\mathbb{K}$ Framework
   Second International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011, Madrid, Spain, May 2011, Revised selected papers,
   editors Ricardo Peña, Marko van Eekelen, and Olha Shkaravska,
   volume 7177 of Lecture Notes in Computer Science pages 90–107, Springer, 2011,
   ISBN 978–3–642–32494–9.

4. Mihail Asăvoae:
   $\mathbb{K}$ Semantics for Assembly Languages: A Case Study
   Submitted to The K Workshop 2011

Technical reports and extended abstracts directly related with the dissertation

1. Mihail Asăvoae and Dorel Lucanu
   Formal Executable Semantics for Timing Analysis
   Technical Report TR SIC-08/11, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación, pages 64–78,
   `federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-8-11.pdf`

2. Mihail Asăvoae:
   A $\mathbb{K}$-Based Methodology for Modular Design of Embedded Systems
   Extended abstract in pre-proceedings of WADT 2012,
   Technical Report TR–08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación, pages 16–17,
   `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`

Publications co-related with the dissertation

Revised papers:

1. Irina Măriuca Asăvoae, and Mihail Asăvoae:
   Collecting Semantics under Predicate Abstraction in the $\mathbb{K}$ Framework,
   Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers, editor P.C. Olveczky, volume 6381 of Lecture Notes in Computer Science, pages 123–139, Springer, 2010,
   ISBN 978–3–642–16309–8.

2. Irina Măriuca Asăvoae, Mihail Asăvoae, and Dorel Lucanu:
   Path Directed Symbolic Execution in the $\mathbb{K}$ Framework
   12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010, Timisoara, Romania, 23-26 September 2010, editors

Tetsuo Ida, Viorel Negru, Tudor Jebelean, Dana Petcu, Stephen M. Watt and Daniela Zaharie, pages 133–141, IEEE Computer Society, 2010,
ISBN 978–0–7695–4324–6.

Others:

1. Adrián Riesco, Irina Măriuca Asăvoae, and Mihail Asăvoae:
   A Generic Program Slicing Technique based on Language Definitions
   Extended abstract in pre-proceedings of WADT 2012,
   Technical Report TR–08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación, pages 91–92,
   `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`

Acknowledgements

to be completed

This work was supported by the the European Social Fund in Romania, under the responsibility of the Managing Authority for the Sectoral Operational Programme for Human Resources Development 2007-2013 [grant POSDRU/88/1.5/S/47646] and by Contract ANCS POS CCE, O2.1.2, ID nr. 602/12516, ctr.nr 161/15.06.2010, DAK.

Table of Contents

List of Figures

Chapter 1

Introduction

Design and analysis of embedded software systems often require careful considerations, on one side, of their correct behavior and on another side, with respect to satisfying certain constraints on resources (i.e. execution time spent, battery consumed, heat dissipated, memory utilized etc). Out of these constraints, the timing behavior of an embedded program is important, due to its interaction with the external environment, and one of the timing-related problems is to compute a priori tight upper bounds on the worst case execution time (WCET) [113] of hard real-time software components of the system.

One may distinguish between two different approaches for the WCET estimation problem: the high-level and the low-level analyses. Used mostly in the early stages of the embedded systems design, a high-level analysis provides the results without considering the architecture description of the system. Its main application is in the hardware/software co-design, where the designer may use the WCET information to decide which components should be implemented in hardware. Low-level analysis considers both the program, usually at the assembly language level, and a model of the underlying architecture. The results of a low-level analysis are used in schedulability analysis and should be accurate to ensure that the timing requirements are satisfied.

Thus, two important issues should be addressed: finding the longest path of the program and the micro-architecture behavior modeling. The longest path analysis returns the sequence of instructions that will be executed in the worst case scenario. The micro-architecture modeling describes the hardware system that the program is executed on. The WCET estimation of a program connects the results of the longest path analysis with the processor behavior analysis.

In this thesis, when we refer to the problem and the solutions for the WCET analysis we mean the ones for the low-level WCET analysis.

We propose a general methodology for worst-case execution time (WCET) analysis centered around a formal executable semantics of the underlying language. We assert that the formal definition of a language has all the necessary information to be used for WCET program analysis and verification. We use the $\mathbb{K}$ rewrite-based semantic framework [101, 29] to define a formal executable semantics for a MIPS-based assembly language, called PISA in the Simplescalar toolset [18]. The choice of the Simplescalar toolset is inspired from [74] and is motivated mainly because it uses a modified `gcc` compiler to generate PISA executables. We call our $\mathbb{K}$ definition of this language *SSRISC* and we refer it as such from this point on. This architecture simulator tool takes C programs and executes them on a configurable architecture, using convenient parameters for cache memories, pipelines or hardware speculation mechanisms.

We advocate on using our formal definition of *SSRISC* as a core, trusted module in a semantics-based WCET analyzer. For example, the execution flow of C programs in our timing analyzer tool would still be: first compile them into executables, then extract assembly programs from them using the Simplescalar disassembler [18], then *execute semantically* the resulting assembly programs in our $\mathbb{K}$ language semantics. $\mathbb{K}$ is highly-modular, allowing us to start with a high-level semantics of the language and then plugging in the $\mathbb{K}$ description of various micro-architecture such as caches, pipelines or hardware speculation techniques. This way, we specialize the original high-level semantics to one specific to a particular processor, which is what we eventually analyze. The $\mathbb{K}$ tool suite (`http://k-framework.org`) provides support for concrete and symbolic execution, for state space exploration of concurrent and/or non-deterministic programs, for LTL model-checking, and for full-fledged verification (see also [100]).

The formal definition of *SSRISC* is designed and implemented in a polymorphic way, to accommodate various abstractions for timing analysis purposes. Then, by "semantically executing" a program we mean (1) a concrete execution, when the initial program state is ground as well as (2) an abstract execution, when the initial program state is underspecified or contains explicit symbolic values. We elaborate more on these in subsequent chapters (3, 6 and 7).

Next, we present a joint overview of the WCET analysis problem and of our semantics-based WCET estimation methodology, via a benchmark program, introduced in Park's thesis [94] and called `check_data`. The program, in Figure 1.1 checks for a given integer array `data` if it contains only positive values. In this case, the `while` statement guarded by the `morecheck` condition terminates and the function returns value 1. Otherwise, if one of the array elements is negative, the function returns value 0.

To detect the longest execution of this program, regardless of the initial values of the array elements of `data`, one has several solutions of choice. We present two such approaches towards WCET estimation. A first choice is to represent the program as a transition system and then perform model checking, as in [87]. Another choice is to rely on integer linear programming (ILP) constraints to express the executions of the program. Since our proposed design adheres to the latter, we keep aside, for now the WCET estimation via model-checking techniques and we elaborate on the ILP-based methodology.

Intuitively, the longest execution path of `check_data` program is when the input array `data` has only positive elements. In the ILP formulation, the longest execution path is the (maximum of) sum of each instruction's contribution to the global time value. This

```
check_data() {

    #define DATASIZE 10
    int data [DATASIZE];
    int i, morecheck, wrongone;
1:  morecheck = 1; i = 0; wrongone = -1;
2:    while (morecheck) {
3:      if (data[i] < 0) {
4:        wrongone = i; morecheck = 0;
        }
        else
5:      if (++i >= 10)
6:          morecheck = 0;
     }
7:    if (wrongone >= 0)
8:      return 0;
      else
9:      return 1;
}
```

Figure 1.1: the `check_data` benchmark program

contribution is expressed as the product between the instruction count and its execution time value. Then, in this example, the instructions at program point `1` are executed once, the `while` condition, at program point `2`, is executed 11 times, the `if` condition, at program point `7` is executed once etc. The ILP representation of the program consists of two types of constraints: structural and functional. For example, since the condition at program point `7` is executed once, it means that only one of the instructions at `8` and `9` is executed, once. This kind of information is represented by the following, informally expressed, ILP structural constraint: the number of time the program flow exits `7` is equal to the sum of the number of times `8` and `9` are taken. An example of a functional constraint is to set loop bounds, say that the number of times the `while` condition at `2` is less than 10. Structural constraints can be automatically extracted from the program, the functional ones are usually manually given or produced as results of data analyses. Both types of constraints form an integer linear programming representation of the target program, which is further sent to an ILP solver. The existing approaches for WCET

```
check_data()
. . .
004002b8 <check_data+0xc8> addu $3,$0,$2
004002c0 <check_data+0xd0> sw $3,0($30)
004002c8 <check_data+0xd8> slti $2,$3,10
004002d0 <check_data+0xe0> bne $2,$0,004002e0 <check_data+0xf0>
004002d8 <check_data+0xe8> sw $0,4($30)
004002e0 <check_data+0xf0> j 00400230 <check_data+0x40>
004002e8 <check_data+0xf8> lw $2,8($30)
004002f0 <check_data+0x100> bltz $2,00400310 <check_data+0x120>
004002f8 <check_data+0x108> addu $2,$0,$0
00400300 <check_data+0x110> j 00400320 <check_data+0x130>
00400308 <check_data+0x118> j 00400320 <check_data+0x130>
00400310 <check_data+0x120> addiu $2,$0,1
. . .
```

Figure 1.2: raw version of the disassembled `check_data` function (partial)

analysis, based on the ILP representation of the program's execution rely on ad-hoc encoding of the programming language semantics.

We propose a different take on the WCET analysis. The Simplescalar tool compiles C code and runs the executables on a configurable simulator. We define, in $\mathbb{K}$, a formal executable semantics for the *SSRISC* language, described in Chapter 4 (in our case it is the assembly language found in the disassembled executables generated through the Simplescalar toolset). This executable semantics compiles into rewriting logic theories, using the $\mathbb{K}$ framework implementation. A snapshot of the *SSRISC* code for the `check_data` benchmark program is in Figure 1.2 and could be formally executed using the $\mathbb{K}$ definition of it. The only required information to execute (in concrete) this *SSRISC* program is to create, automatically, or to give, manually, the initial configuration of the program, with ground values. We investigate both possibilities in the subsequent Chapter 4, where the *SSRISC* semantics is described. Our further goal towards WCET analysis is to consider this concrete execution of a given program, on a specified architecture, informally described next, as the basis for an abstractization process.

The low-level WCET analysis considers two sub-problems. The first is with respect

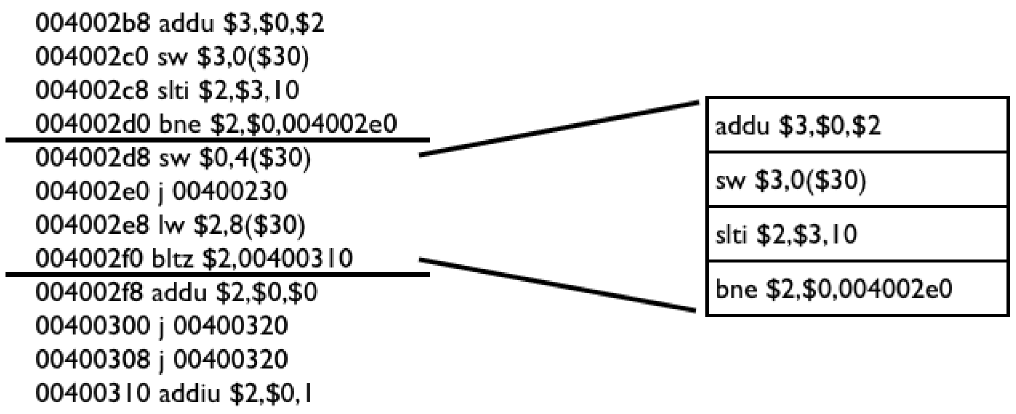


Figure 1.3: Instruction cache placement of a fragment of a *SSRISC* program

to the program's execution paths - which executions are feasible or infeasible, while the second follows how the micro-architecture elements influence the timing behavior of the program, at the level of instructions. If we consider that the *SSRISC* representation for the program `check_data`, in Figure 1.2, is executed in the presence of an instruction cache, then an instruction contribution to the global timing behavior of the program depends if the instruction fetch results in a cache hit or miss. For example, let us assume that the jump instruction at the (symbolic address) `check_data+0xf0` is executed four times, along the (estimated) longest execution path, and one time results in an instruction cache miss and the rest in cache hits. The total contribution time for this particular instruction is three times the normal execution time plus one time the penalty due the cache miss. Hence, the previously ILP formulation for the WCET estimation is refined, to take into consideration the micro-architecture behavior. An example of a fragment of the `check_data` program, mapped to a fully-associative instruction cache is shown in Figure 1.3.

The complexity of the underlying architecture elements varies, with the cache memories, pipelines and hardware speculation being the most popular in the timing analysis field. In Chapter 5, we propose a modular and parametric modeling for instruction and data caches, taking into account various cache organizational information, replacement policies, writing policies etc. All these together with the formal executable semantics of *SSRISC* allow us to execute a concrete program on a concrete, maybe partially specified, architecture. This sets the basis for further abstraction encodings, directly over this combined core, and we treat such definitions in Chapter 6 and Chapter 7. We elaborate on the general idea next.

The $\mathbb{K}$ framework, through the explicit configuration manipulation, allows to define abstract executions, just like the concrete one, for a particular system. The two sub-problems of WCET analysis - the path analysis and the processor behavior analysis - emphasize on the necessity of defining abstract executions on both the program flow and data. We approach these from a novel perspective, by trying to execute the formal semantics of *SSRISC* and to retain just enough information for accurate results of the analysis.

A starting point into investigating definitions for program analyses is to obtain the control flow graph of a program. We rely on the configuration abstraction [54] of the $\mathbb{K}$ framework to "isolate" the control-related information of the language configuration, this happening during the execution of the target program on the formal semantics. We employ a similar strategy when we consider the ILP-based encoding of the program - the first half of the standard ILP + AI method (integer linear programming + abstract interpretation). Once such structural ILP constraint, for the *SSRISC* program in Figure 1.2: the number of times the instruction `bltz` at program address `004002f0` is executed is equal to the number of times the program flow exits it, and is equal to the number of times the program flow enters the subsequent program counters: `004002f8` (fall-through case) and `00400310` (jump case).

The other half of the ILP + AI approach, the AI part, refers to the context of processor behavior analysis. It is designed and implemented in a similar fashion as the ILP formulation. The two abstractions, while different in nature, use a common principle of development - abstraction over concrete description of the system. We discuss more on this at the end of this section. Also, with respect to control-flow abstractions, we propose a simple annotation language for the *SSRISC* language, to accommodate ILP functional constraints, as loop bounds, for particular, common, types of loops. The flow-based abstractions are presented in Chapter 6.

The result of the WCET analysis should be safe and tight. While safeness is ensured via abstract interpretation based verification methods, the tightness of the estimation involves an important subproblem, that of infeasible path detection. By definition, an program execution path is infeasible if it cannot be reproduced under any possible input. An undetected infeasible path could be the cause of a large overestimation for the WCET. It is necessary to perform data analyses to find and eliminate, as much as possible, the infeasible execution paths of a program. Data analysis has another important application for timing estimation: in the context of data cache behavior prediction, the program's data results in cache hits and misses. Therefore, a similar refinement of the ILP formulation of program uses data, classified with respect to the interaction with data caches.

We present in Chapter 7 two simple data abstractions. The first considers a special symbolic value, supported through subsorting, by the formal executable semantics of *SSRISC*. For example, this abstraction, supported by a flat lattice, allows to symbolically execute the program `check_data`, in Figure 1.2. The branch instruction at program point `check_data+0x100`, under the assumption that register `$3` has a symbolic value, has both branches enabled. The second is an interval analysis that is useful to characterize memory addresses as intervals. Its main applicability is in the context of data cache behavior prediction. We design and implement an interval analysis in a similar fashion as the flow-based analyses counterpart - directly over the formal executable semantics.

Up to this point, we overviewed one of the most successful methodologies for WCET analysis - the ILP + AI approach. Since we work in the $\mathbb{K}$ framework, and through its current implementation, a $\mathbb{K}$ definition compiles into a specification in the Maude system, $\mathbb{K}$ indirectly offers access to all Maude's support tools. Out of these, there are two search-space exploration methods: reachability analysis using a specialized command called `search` and an LTL model checker which, as its name points, offers support to full LTL model checking. Our $\mathbb{K}$ formal definition of *SSRISC* can be model checked, in this way, for timing properties. We use model checking techniques as an alternative for our main approach that integrates abstractions over the concrete specification. We exemplify, in Chapter 6, how to utilize the reachability analysis offered by Maude when we apply it to construct the control flow graph of a program. In the data analysis department, we again use the `search` command to detect a special case of infeasible paths. For example, a *SSRISC* program that has division instructions, is susceptible of raising division by zero errors (similarly, if the program contains signed operations that may raise overflow errors). These kind of errors are arithmetic and are usually removed before performing timing analysis, using either specialized data analyses on the high-level code or just assume that numerical errors do not occur. Using reachability analysis on the *SSRISC* language definition in the $\mathbb{K}$ framework, the assumption that timing analysis has to be performed on an error-free code, is relaxed. The intuition behind this is that the formal executable semantics has all the necessary information, including arithmetic checks, to execute programs.

The overall system, presented in Chapter 3 is built around the $\mathbb{K}$ framework definition of the *SSRISC* assembly language and micro-architecture modeling. The system is modular and parametric, allowing, in this way to execute programs on various architectures, or in various degree of abstractizations. The novelty of this design is that it takes advantage of the "separation of concerns", promoted by $\mathbb{K}$. We propose two kind of extensions to the concrete system. First it is an extension at the level of data manipulated by the *SSRISC* definition, which can be concrete, symbolic, interval-based values etc. Second, it is an extension at the level of $\mathbb{K}$ modules to integrate abstractions, relying on the configuration abstraction mechanism of the framework. This results in execution as an interleaving of abstract and concrete execution steps. If we strictly refer to the abstract interpretation based analyses, the seamless integration with the concrete specification of the system enjoys several important properties. Briefly, the concrete specification is not really concrete as it is able to symbolically execute the programs on particular architectures, with respect to a symbolic value domain. A second point is that the state-based abstractions are more inclined to be integrated. A third point is that, when the execution "returns" from the abstract domain, it should not pick a representative from the abstract state (represented as a set of concrete states), only to return with the entire abstract state. This methodology of integrating various kinds of abstractions in the concrete system is by a meta-algorithm that exploits the constituents (modules, dependencies between modules etc) of the overall system.

The WCET analyzer was implemented in the $\mathbb{K}$ framework and tested on various kinds of programs. The concrete implementation relies on testing at the level of each unit, (i.e. instructions, architecture elements etc) or testing with hand-made programs while for the abstract implementation, we use a standard set of benchmarks, from Mälardalen University [47]. The current prototype implementation could be extended in several directions, which are overviewed in the last chapter of this thesis.

1.1 Motivation

Ideally, program analysis tools should be based on rigorous semantics of the employed programming languages. Unfortunately, giving a formal semantics (using conventional approaches) to a real language is a non-trivial matter; moreover, even when a semantics is available, it is often not easy to use it for program analysis. Recent research in rewriting logic semantics and in tool development based on such semantics [101, 29] shows encouraging results with respect to both expressiveness and scalability. Moreover, the application of these techniques in the context of real-world low-level languages such as Verilog [81] gives us hope that the theoretically ideal semantics-based approach to program analysis may be, after all, also practically feasible.

We decide to explore the expressiveness and the scalability issues when we propose the design and implementation of a WCET analyzer. A particularity of this approach is to use a formal executable semantics as the basis for developing analysis and verification methods. Since we follow the $\mathbb{K}$ framework desiderate of separation of concerns, we investigate to what extent (and for what kind of abstract executions) the concrete semantics could be used, as it is, during the analysis process. Informally, this view of building a system over a reliable, trusted kernel corresponds to the generic approach used in theorem provers, for example in the PVS system [106]. The kernel - in our case the formal executable semantics of the assembly language + micro-architecture description, is considered trusted with respect to extensive testing. The system is designed with several clear objectives in mind, as presented in the next subsection, and it is stretched when we integrate one of the most successful methodologies in WCET analysis - the ILP + AI approach.

1.2 Objectives

To summarize our goal in one simple sentence: we propose a definitional, rewrite-based WCET analyzer, at both the level of design and the implementation. Before we provide additional insights into our decisions/objectives, we present the two points of view that lead to our solutions.

- Embedded software systems pose a dual nature, with the external environment being their counterpart. The interaction between these two components, burden the embedded program to satisfy certain functional and non-functional constraints (i.e. time spent, memory space occupied, power consumed, heat dissipated etc). The problem of timing analysis, and in particular the estimation of a WCET for a given program is important in the schedulability analysis, performed in complex, safety critical systems.

- A program in execution is a sequence of modifications, applied over a common structure, named the program configuration. We adhere to the following view - computation as rewriting - where each modification is capture via one or more rewriting steps. Then, the rewriting logic project provides the formalization and the necessary tools, through the Maude system, to facilitate, in a declarative style, the development of system specifications. The $\mathbb{K}$ framework is especially designed to programming languages definitions and their afferent analysis and verification technologies. $\mathbb{K}$ provides a concise notation and a new execution style, that exploits more concurrency during the rewriting process. What is of particular interest to us is that the execution of a rewriting-based specification, be it a program or a architecture description, is correct with respect to that particular specification.

Now, merging the two points, the system we propose addresses the problem of WCET analysis from a novel point of view, taking advantage of the underlying technology that the rewriting-based environment provides: executability and modularity. We decide to design the WCET analyzer around a trusted core - a formal semantics of a programming language of interest and a concrete description of the micro-architecture elements - and to extend it, via a configuration abstraction mechanism, to allow abstract executions. The key is to keep the trusted core unmodified (by trusted we mean a well-tested specification) This would ensure further specialized analyses to be integrated in a smooth, controlled manner, achieving in this way a proper separation of concerns.

Our list of sub-objectives include, along with exploiting the executability and modularity of $\mathbb{K}$, parameterization and reusability. We elaborate a bit on each of these four desiderates and how they could be identified in our definitional WCET analyzer.

Executability

The $\mathbb{K}$ framework provides executability, which allows to test the specification and to get correctness guarantees about it - a $\mathbb{K}$ run of a specification is correct with respect to that particular specification. The formal semantics of the MIPS-based assembly language and the architecture description could be concretely executed and, using test cases, made it trusted.

Modularity

The $\mathbb{K}$ framework, through the rewriting logic, provides also the modularity of the analyzer. Since we basically reason about the execution of a program on a specified architecture, it is convenient to keep the models of these two components separately. From a structural point of view, $\mathbb{K}$ uses modules to enclose the necessary information to specify a particular system (or concept). From a functional point of view, these $\mathbb{K}$ modules communicate between among each other using specialized terms, which act like messages.

Parameterization

The part of the system that deals with the micro-architecture modeling is designed and implemented to cater for a number of hardware-specific arguments. In our particular case, we refer to a parametric implementation for cache memories behavior (i.e. cache sizes, cache organization, replacement policies etc). This parameterization is present in the generic hardware simulators, useful mostly to experiment with the execution of programs in various environments. In our case, this parameterization is also useful when we encode cache-specific behavior analyses, and to obtain, in this way, a family of abstract executions.

Reusability

One desiderate of our design is to consider a core specification, deemed as being trusted and to extend it with additional functionality, mostly oriented towards analysis and verification. Then, from this point of view, of the relationship concrete-abstract executions, we argue that the implementation is reusable. A direct consequence is to simplify the process of implementing abstractions, directly over a concrete or another abstract specification of a system.

1.3 Contributions

Towards achieving our objectives, the key contributions of this thesis are:

1. **Formal Definition of a RISC Assembly language**

   We design and implement, in $\mathbb{K}$ a complete formal definition for a MIPS-like assembly language — the PISA instruction set from the Simplescalar toolset. This is novel with respect to the existing language definitions in rewriting logic and the $\mathbb{K}$ framework. From this definition, we could extract several useful subsets, such as the integer-based subset of the language or the representation for the disassembled executables, the latter being the focal point in our further developments.

2. $\mathbb{K}$ **Specifications for Split-Caches**

   We design and implement a modular and parametric $\mathbb{K}$ specification for instruction and data cache memories behavior. The parametric aspects refer to both general caching (i.e. cache size, associativity, replacement policies on read, look-up) as well as particular to family of caches (writing policies in data caches). This is new with respect to the existing architecture specifications in rewrite-based programming environments. The architecture part is designed to accommodate further extensions for timing analysis.

3. **General Methodology for Integration of Abstractions**

   We propose a general methodology for integrating abstractions, directly over the concrete specification of the system, consisting of the programming language definition and the micro-architecture modeling. This methodology is presented as a meta-algorithm that is instantiated for each abstraction of interest, by following specific design steps. The consequence is that, during its execution, the abstract steps are interleaved with the concrete steps. In this way, we could define and measure a reusability factor. Instances of this approach - Contribution 4 and Contribution 5 - have been tested on a set of standard benchmarks for WCET estimation. This methodology is new and $\mathbb{K}$ specific, as it takes advantage of a configuration abstraction mechanism provided by the $\mathbb{K}$ framework.

4. **Control Flow Abstractions in** $\mathbb{K}$

   We design and implement, in $\mathbb{K}$, two standard flow-oriented abstractions for timing analysis: an abstraction for control-flow graph (CFG) extraction and an abstraction for generation structural integer linear programming (ILP) constraints. The first analysis enjoys two variants: one which considers minor modifications of the original semantics and relies on reachability analysis to extract the CFG edges and another which keeps the core language semantics unmodified and interleaves concrete and abstract execution steps. The second analysis is part of the general methodology of ILP + AI for WCET analysis and extracts a particular kind of structural information from the program. This is new with respect to a general way of defining abstractions, using the general configuration abstraction of the $\mathbb{K}$ framework.

5. **Cache Behavior Abstractions in** $\mathbb{K}$

   We design and implement, in $\mathbb{K}$, a family of standard analyses for instruction cache behavior. These analyses - may, must and persistence, are used to classify the program's instructions with respect to their interaction with the caching system. Their definition is an instance of the previously mentioned meta-algorithm - Contribution 3.

All these set the base for the first definitional WCET analyzer.

1.4 Thesis Organization

Our design and implementation of the semantics-centric WCET analyzer starts with an overview of the design methodology, modular and executable. Next, we introduce the kernel of our approach - the formal executable semantics of the assembly language of interest and then, the description of the micro-architecture elements, which is particularly important in the low-level WCET estimation process. Two families of analyses, on program flow and on program data are treated separately, with an emphasis on path analysis, respectively on processor behavior analysis. We conclude and present future directions of research. This thesis' chapters treat all the aforementioned topics, in the specified order.

Chapter 2. Preliminary Notions

Our approach towards WCET analysis is based on the $\mathbb{K}$ framework, a programming language definitional framework, originated from the rewriting logic semantic project. General notions from these two frameworks set the working environment for our further design and implementation. The specifics of WCET analysis problem and its solutions require a wide range of techniques from integer linear programming to abstract interpretation and model checking.

Chapter 3. Modular System

We propose a modular design, taking advantage of the intrinsic characteristics of a multifarious $\mathbb{K}$ framework based definition. This chapter investigates this design from two aspects: one with respect to the overall system and the other referring to specialized components (i.e. builtins, the main memory system etc). The design methodology is general and allows both concrete and abstract extensions of the system. We choose to present this approach as a meta-algorithm, which will be instantiated in the subsequent chapters.

Chapter 4. $\mathbb{K}$ Definition of *SSRISC*

The central piece of this work is a formal $\mathbb{K}$ definition (syntax and semantics) of a MIPS-based assembly language, which is part of the specialized toolset, called Simplescalar. The Simplescalar simulator permits the executions of the compiled C programs under various, parameter-specified architectures. We are interested to perform WCET analysis directly over the disassembled executables and use as input such low-level programs. The given formal definition executes, apart from the disassembled programs, more traditional assembly language programs - traditional in the sense of label usage as symbolic memory addresses.

Chapter 5. Micro-Architecture Modeling in $\mathbb{K}$

The execution of embedded software systems is greatly influenced by the underlying architecture. When the main interest is on the non-functional requirements of such program executions, it becomes rather mandatory to include, in any kind of quantitative estimation, the architecture's behavior. In the context of timing analysis, the most studied topic, with respect to micro-architecture elements, is the impact of the instruction and data cache memories. We propose a parametric implementation of a family of cache memories, taking into consideration both structural-specifics (cache size, cache associativity) and functional-specifics (block identification and replacements policies).

Chapter 6. Control-Flow Abstractions for WCET Analysis

To facilitate giving abstractions, the program is usually represented at a level of a special structure, called control flow graph (CFG). The problem of identifying the CFG of an assembly program is a difficult problem, one of the motives is the presence of indirect jumps. We present a model-checking solution, via the underlying technology provided by the Maude system. A similar, control-flow abstraction is to extract a particular kind of flow information as integer linear programming (ILP) constraints. Next, we show how our modular design accommodates an encoding of an abstract interpretation (AI) based analysis to classify instructions with respect to the instruction cache behavior. These two aspects form the well-known approach for WCET analysis - the ILP + AI methodology. We measure a reusability factor, the percentage of concrete part used during an abstract execution of the system.

Chapter 7. Data-Flow Abstractions for WCET Analysis

A tight estimation for the WCET of a program relies on the ability of a particular abstraction to detect and eliminate infeasible paths. Since the formal executable semantics of a programming language contains all the necessary information to encode abstractions over it, it could be used to detect a particular type of infeasible paths, such as erroneous paths. We present a simple data abstraction that "learns" new information in the branching points of the program. Also, we present a rewrite-based formulation of the most used data analysis to classify program information with respect to the data cache behavior.

Chapter 9. Conclusions and Future Work

The last chapter proposes a set of interesting examples on how this work can be extended. Taking advantage of the inherited executability of the $\mathbb{K}$ framework, an interesting direction of research would be to execute the micro-architecture code, which is usually described as a hardware description language (HDL). Another immediate extensions should be to improve over infeasible path detection, as well as extending to include other micro-architecture features such as the pipelines. We end with the conclusions.

Chapter 2

Preliminary Notions

In this chapter, we overview some of important background information that we use in this work. We present the concepts mostly from the perspective of their applicability to the WCET analysis and of the specification with the rewriting logic and the $\mathbb{K}$ framework.

We organize our section of the preliminaries in the following way: we start with the underlying technology that we use, meaning the rewriting logic and the $\mathbb{K}$ framework, then we cover the timing analysis specific notions on both the modeling, meaning the integer linear programming, and verification techniques, meaning the abstract interpretation and the model checking frameworks.

2.1 Rewriting Logic

Rewriting logic [83, 79] integrates computational and logical aspects, both are necessary to define and reason about a large range of applications from concurrent systems, programming languages semantics, software and hardware modeling languages as well as system reasoning tools. Before we quickly overview the framework of rewriting logic, and without being exhaustive, we enumerate several important research achievements on the aforementioned application domains (and important throughout this work):

- concurrency: generally presented in [84] and an important application on graph rewriting [104]
- programming languages semantics : Java in [39] and JVM in [40], a more general view on structural operational semantics (SOS) embedding in the rewriting logic framework [105] and the rewriting logic semantics project in [86]
- hardware systems and modeling languages : processor specification and verification in [51] and [32], hardware description langauges (HDLs) - Verilog in [82] and ABEL in [62], asynchronous circuits in [61]

When such complex systems are formally specified, they impose static and/or dynamic modeling aspects, and the rewriting logic framework provides the meanings to capture them. A rewriting theory consists of an equational theory and a set of rewrite rules (possibly conditional). An equational theory is represented by a system's signature used the state constructions and by a set of equations to establish equivalence relationships between the states. The equational theory addresses the static aspect of the modeling. The rewriting rules of a rewriting theory represent the concurrent transitions between the states specified by the equations. Each rewrite rule $l \to r$ transforms the left hand side pattern l into the right hand side r. Because a rewriting logic specification is an abstract transition system (with the equations determining the states and the rewrite rules, the transitions between these states), controlling the sets of equations and rewrite rules, one basically controls the abstraction level of the specification.

The Maude system [24] is the implementation of rewriting logic and, together with a number of integrated methodologies and tools such as a reachability states exploration tool, an LTL model checker [36], an inductive theorem prover, as well as other specialized checkers [34, 35], it enables specification and analysis of programming languages. With respect to what we stated in the previous paragraph, a rewriting logic definition of a system, in Maude, helps to simulate the system's behavior. One of the Maude's strengths is the possibility of using reflection to achieve meta-level computation [23].

There are several extensions of the Maude system, either to address definitional capabilities, such as Full Maude [33] or to address domain specific areas of research, such as the Real-Time Maude [93]. This tool supports the design and verification of real-time and hybrid system, using a specification mechanism based on rewriting logic and time-based extensions of the Maude's tools (i.e. time-bounded LTL model checking, timed rewriting).

The Maude system provides the runtime environment of our $\mathbb{K}$ specifications: for the *SSRISC* assembly language in Chapter 4, the underlying architecture (i.e. cache memories), in Chapter 5 and the integrated analyses, in Chapters 6 to 8.

2.2 The $\mathbb{K}$ Framework

The $\mathbb{K}$ framework emerged from the rewriting logic semantics project [86] and it is specialized for the design and analysis of programming languages. A $\mathbb{K}$ specification consists of *configurations*, *computations*, and *rules*. The configurations are labeled and nested $\mathbb{K}$-cells and represent the structure of the program states. The rules in $\mathbb{K}$ are of two types: *computational rules* which represent transitions in a program execution, and *structural rules* which provide structural changes of a state in order to enable the application of computational rules. The computations in $\mathbb{K}$ are automatically produced based on the configuration and rules specification, and represent executions of the specified system. The $\mathbb{K}$ framework allows modular and executable definitions of programming language semantics.

The $\mathbb{K}$ framework, described in [101], supports the definitions of programming languages using a specialized notation for manipulating program configurations. $\mathbb{K}$ shows its versatily when handling definitions of real languages, such as C in [37] or Verilog in [81] as well as definitions for type systems, model checkers or a Hoare style program verifier [100]. K-Maude tool [29] implements the $\mathbb{K}$ framework on top of Maude system and provides, in this way, access to all Maude's aforementioned supporting tools.

A language defined in the $\mathbb{K}$ framework follows several steps. One starts with the language (abstract) syntax, which is based on the expressive context-free style (using data constructs like lists of maps without their a priori definition). The second step is crucial to the language semantics definition - to establish what is the language configuration. Configurations in $\mathbb{K}$ consist of all the semantic entities to represent any of the program's executions and, constructively, are defined as nested structures of semantic constructs (the $\mathbb{K}$ cells). The third (and last) step is the definition of the language semantics. The executability characteristic of the semantics relies on the special k cell which consumes tasks from a list of computational tasks, in a continuation-based style.

The $\mathbb{K}$ notation is based on the matching supported by the Maude system, making it concise and easy to use. Currently, there are three representation styles of the $\mathbb{K}$ configurations, to address different application domains [104]: bubble, mathematical and ASCII representations. Throughout this work, we adopt the followings: the ASCII notation to represent the $\mathbb{K}$ code snippets and the mathematical notation to represent the configuration and the meta rules (i.e. a pseudo-code using the $\mathbb{K}$ style).

We rely on the $\mathbb{K}$ framework strengths - notation and compilation - to define various systems (i.e. programming languages, architectures, program analyses etc) and therefore, $\mathbb{K}$ is omnipresent throughout this dissertation.

2.3 Integer-Linear Programming

Mathematical based modeling is a popular approach in the area of embedded and real-time systems, where such systems or only parts of them are modeled using numerical variables. A prominent method that helps to optimize cost functions expressed using such variables is linear programming [21]. Obviously, the definite characteristic of linear programming is that a model, also called a linear program, is represented as linear equality and linear inequality constraints. Solving a linear program means to determine the solution (i.e. a particular smallest of largest value) in the solution space, which is represented as a convex polyhedron.

A linear programming problem where the modeling variables are integers is called a integer linear programming (ILP). Modeling with integer variables could cover a wide range of applications, from logical requirements to structural and functional requirements in the program. To be more specific, we mention the applications of ILP to the embedded systems research area:

- in hardware-software co-design of embedded systems [68]
- in exploiting task level parallelism in multiprocessor systems on chip [25]
- in code generation for embedded systems compilers, in [59]
- in energy consumption prediction for wireless sensor networks [58]

With respect to the type of variables' values, ILP has several popular variants: a mixed ILP where some of the modeling variables are integers and some are not and a ILP 0-1 case, with the values restricted to these two values. In this work we focus on the standard application of ILP to embedded software systems, with the variables having only integer values. For more general information on ILP and its application domains, we refer the reader to [57].

We present next how integer linear programming addresses various modeling issues in the context of program analysis and verification. We restrict our ILP overview to its application to timing analysis.

In [75], the authors model the control flow graph as a integer linear program, the so-called implicit path enumeration solution. A problem formulated using ILP consists of two parts: defining a cost function and deriving constraints on the variables used in

```
1.   add r1, r2, r0;
2.   beq r2, r3, 10;
3.   bne r1, r3, 6;
4.   sw r1, 4(r1);
5.   j 8;
6.   add r1, r3, r2;
7.   sw r2, 8(r1);
8.   add r2, r2, r3;
9.   j 2;
10.  sw r1, 4(r2);
```

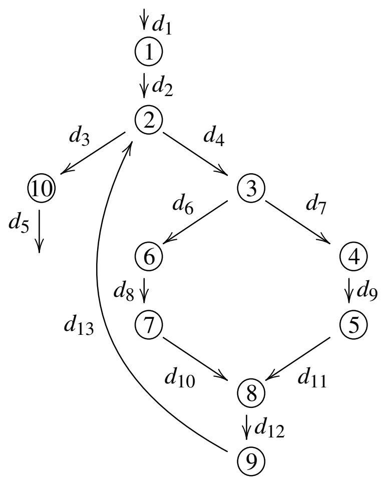


Figure 2.1: Assembly program (left) and its control flow graph (right)

the cost functions. The cost function needs to be maximized and it is usually the number of CPU cycles the program takes when executed. A basic block is the maximal sequence of instructions for which the only entry point is the instruction and the only exit point is the last instruction. Each variable in the cost function represents the execution count of one basic block of a program and is weighted by the execution time of that block. Variables are used accordingly to the traversal of the edges of the control flow graph. The worst case timing for a program is a given by the maximum value of the objective/cost function. Regarding the linear constraints, the objective is to automatically generate them from the control flow graph. However, additional information about the program is required, in cases of loop and recursion bounds, which cannot be inferred automatically.

The ILP modeling is part of the most successful approach in timing analysis, namely the ILP + AI approach. We present the ILP component on the example from [75] adapted for a subset of a RISC assembly language. The ILP-based path analysis requires the modeling of the control flow graph as an integer linear program, rendering the so-called implicit path enumeration solution. The cost function needs to be maximized and it is usually expressed as $\sum_i c_i \times x_i$, where x_i and c_i are, for instruction i, the number of executions and respectively the cost of one execution. The objective is to automatically generate linear constraints for the variables x_i from the control flow graph. For the program in Fig. 2.1, the constraints $x_1 = d_1 = d_2$, $x_2 = d_2 + d_{13} = d_3 + d_4$ or $x_7 = d_{10} + d_{11} = d_{12}$ are such examples, where d_i denotes the traversal count of the labeled edge. For programs with loops and recursive calls, additional information (i.e. bounds on the number of iterations/calls) should be provided or derived.

These structural and functional constraints, together with the objective function form the integer linear problem, which encodes the program under analysis. In this work, we use a non-commercial ILP solver, called `lp_solve` [10]. We elaborate on how to integrate both the ILP modeling of programs and the solver in Chapter 8.

2.4 Abstract Interpretation

Abstract interpretation was introduced in [26] and since then, it established itself as one of the major program reasoning techniques, along with model checking and deductive verification. In general, a program's set of possible executions cannot be explored in an exhaustive manner, therefore, a simplification of this search space is required. Inevitably, this simplification - called abstraction - induces a loss of information.

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 [115].

We start with the standard representation of the target program as the control flow graph (CFG) with the nodes - the program points and the edges - the flow between these points. These programs are executed based on the programming language semantics. Such a semantics is a triple consisting of a set of program states, a set of initial states and a transfer function that defines the program state update for each control flow edge. 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 does what its name imply, maps/represents concrete to abstract states. 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.

The representation function is not unique, there are actually a set of possible representations of a concrete state with respect to a domain. 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. Another 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.

The abstract interpretation based program analyses are used to a wide range of applications. We mention next only several of them, related to the embedded and real-time application domains:

- various memory related analyses for the heap in [19], the stack in [98] or memory consumption prediction in [88]

- control flow graph extraction in [65]
- micro-architecture models for embedded systems such as the pipelines in [103] and the cache memories in [41]
- program transformation with application to software security in [30]
- the ASTREE tool [28] checks runtime errors on large safety critical systems.

We elaborate on Chapter 7 and Chapter 8 on direct applications of abstract interpretation on timing analysis, such as data analyses and respectively cache behavior analyses. We refer the reader to [27] for the methodology of systematic design of program analyses and to [11] for a general view on embedded systems verification using abstract interpretation.

2.5 Model Checking

Model checking [22, 97] is the second major verification technique that we discuss, after the abstract interpretation. We aim to present the model checking applications to embedded systems, in general, and to the specific timing analysis area, in particular. We start with a general view on model checking.

The model checking technique verifies, completely automatic, if a given model of a system meets a specification, given as a logical formula. The model is represented as a transition system, a directed graph with nodes representing the states of the system and the edges representing the transitions. Also, a set of atomic propositions are in each node. Model checking applies to finite-state systems, a restriction that is suitable for the domain of WCET analysis, because the programs of interest are a class of hard real-time programs. Actually, the model checking is more often applied to hardware designs than to software systems.

The difficulty in applying model checking techniques is the state explosion problem. Numerous solutions are available, from exploiting the structure of the search space, like the symmetry reduction to changing it via abstractions.

- The first breakthrough to handle the state explosion issue is via the binary decision diagrams (BDDs), described in [16], the model checking using BDDs is called symbolic model checking [17].
- Another solution is the one used by SPIN model checker [56], the partial order reduction (i.e. symmetry of the search space).
- An unsound, but popular method is the bounded model checking [12], where the underlying transition system is unfolded a fixed (i.e the bound) number of times.
- Finally, we mention the abstract model checking techniques, which rely on a simplified system, using convenient abstractions, like in [64, 46, 80].

The embedded systems domain uses model checking for a wide variety of applications such as: on demand instantiation of nondeterministic values in embedded programs in [91], power management in [92], or memory modeling in [44].

We elaborate next on using model checking for WCET estimation. The first use of model checking technology in WCET estimation is introduced in Metzner's paper [87], where a processor with simple cache is checked via multiple runs, selected in a binary search fashion. More recent works are around the UPPAAL model checker [9], where the control flow graph of the program and various micro-architecture features are represented as timed automata. In [78], the focus is on multicore systems with timing information extracted from the TDMA and FCFS memory bus models, whereas [31] proposes modular representations of micro-architecture of several ARM processors. In both papers, UPPAAL explores the timed automata models and the WCET of a program is extracted from their clock constraints.

We apply model checking techniques, via Maude system, for a procedure of control-flow graph (CFG) extraction, in Chapter 6, and for erroneous path detection using the executable semantics, in Chapter 7.

2.6 Processor Behavior

Timing analysis should be feasible, efficient and provide accurate and tight results. All these parameters depend heavily on the underlying architecture that the program under scrutiny runs upon. Modern microprocessor architectures have cache memories (i.e. multi-level caches, split-caches, victim caches), pipelines (out-of-order execution pipelines, deep pipelines), speculation mechanisms (i.e. control and data speculation).

The cache memories are small and fast memories to attenuate the delays incurred during the memory accesses. The pipelines exploits the instruction level parallelism to facilitate overlapping of instruction executions. Speculation is important to predict, during a program execution, various behaviors based on the sequencing of instructions (i.e. guessing what is following instruction after a branching point). All these micro-architecture elements directly influence the aforementioned parameters of a timing analysis. Therefore, as it is stated in [114], an execution time of a program on a given (partial) specification of an architecture ranges from "all good" scenario (i.e. all cache references are hits, no pipeline stalls, no mispredicted instructions etc) to "all bad" scenario (i.e. cache misses, pipeline stalls, only mispredictions etc).

The study of architecture's influence over the timing analysis covers (a snapshot of) the following classes of elements:

- the class of cache memories is the most studied one, in comparing with the other micro-architecture components: instruction caches in [76], data caches in [111], shared caches in multicore processors in [117]

- the pipelines in [69] for in-order execution and in [73] for out-of-order execution

- the branch prediction schemes in [13] and in [72]

The micro-architecture models are integrated with models of the input program, leading to separate analyses for the two components. With this view in mind, the architecture behavior is modeled by computing invariants about the processor behavior at each program point. The presence of invariants allows the derivation of safety properties. One example of such invariant is the set of memory blocks that are in the cache every time the execution reaches this program point. Abstract interpretation [26] has been used to compute invariants about cache and pipeline contents. The notion of collecting semantics is important when doing program analysis via abstract interpretation. The contribution of cache, for example, in WCET calculation is considered by computing a collecting cache semantics [108]. Two analyses are defined, a must analysis that determines the memory blocks resulting in cache hits at a given program point and a may analysis that determines the cache misses.

The processor behavior modeling is presented in Chapter 5 (for the parametric $\mathbb{K}$ definitions for instruction and data caches) and in Chapter 8, where we integrate, in our design, a standard cache behavior analysis.

Chapter 3

A Modular System for Timing Analysis

The increased complexity of current software systems requires, from a programming language definitional point of view, the manipulation of large program configurations. In the context of embedded software systems, it is important, as well, to capture the underlying architecture behavior. We use the $\mathbb{K}$ framework [101] to study not only how to modularly define program executions in complex environments but also how to apply abstractions on them. The informal understanding is of a system with a trusted kernel which can be extended to accommodate new structural and functional features.

In this chapter we present a generic, modular design for embedded systems and then, a particular model (or instance) of it that is the structural core of our WCET timing analyzer. To be more specific, we consider a formal executable semantics of a MIPS-based assembly language called *SSRISC* as the processing core and various memory models (at different levels of abstractions). The modeling methodology is based on two key features of the $\mathbb{K}$ framework: the configuration abstraction and the inherited modularity (from rewriting logic). We start with a general presentation of the overall methodology.

3.1 Overview

Let us consider a complex computational system, specified in $\mathbb{K}$ that has the general configuration, denoted as C_{global} (or $\langle$ *system_config* $\rangle$ in Fig. 3.1), which comprises of all semantic entities that are necessary to capture the system's behavior. Following a top-down approach, we take the C_{global} configuration and split it into a number of sub-configurations: $C_1, C_2, \ldots, C_n$, not necessarily disjoint. There are several important observations with respect to this form of abstraction, called SPLIT. First, at an informal level, each sub-configuration should be used to capture a well-defined component (i.e. $\langle$ *subsystem_i* $\rangle$) 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 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. 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. 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. At the end of this chapter, we elaborate on possible applications of this computational model in the

$\mathbb{K}$ framework.

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 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 called WRAP in Fig. 3.1 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. From this point on we refer to this methodology of abstraction by SPLIT + WRAP.

We present SPLIT + WRAP as a meta-algorithm, in Fig. 3.1, then we show how we create an instance for a WCET analyzer, instance that is used in the remaining chapters of this dissertation.

The methodology of abstraction 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 special 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

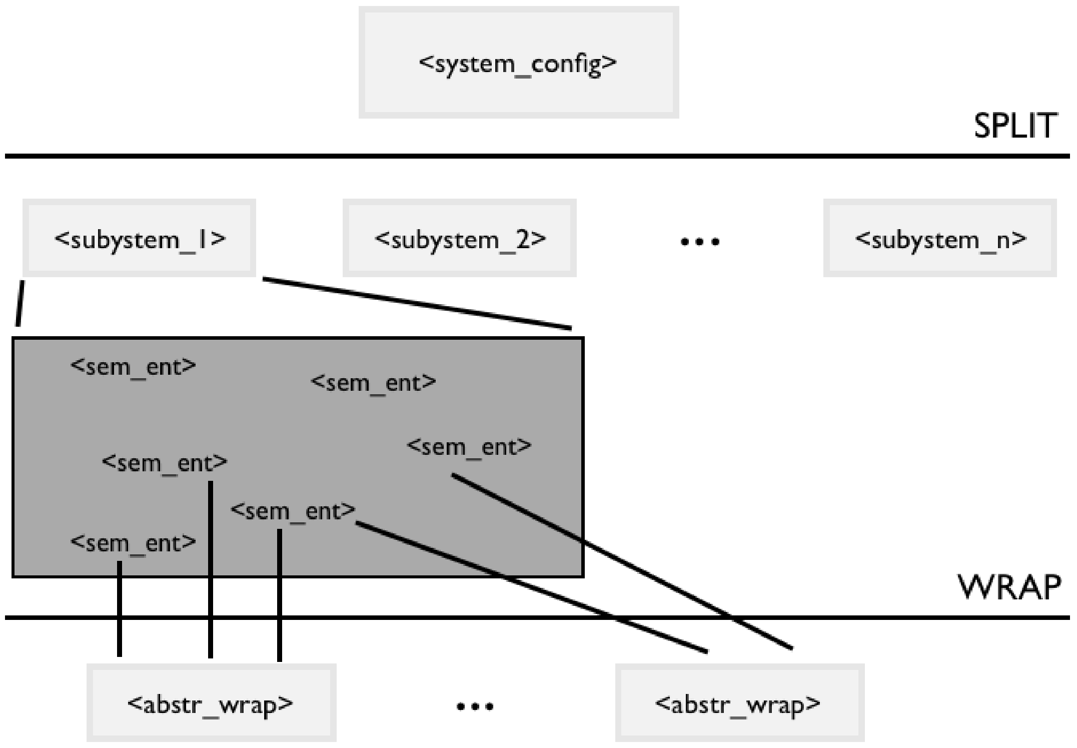


Figure 3.1: General module system - SPLIT and WRAP abstraction styles

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 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 "produces" 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`,

```
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 $M_j$
    <abst> <k> </k> $Rest$ </abst> - the abstract configuration, with $Rest$
                                              being abstraction specific
    <concr> </concr> - the wrapped concrete configuration

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

// INIT

  $S \leftarrow S \cup$ {<abst> <k> ($init$ => $busy$ </k> $Rest$ </abst>
            (. => <concr> </k> $C_{1,i}(init)$ </k> $Rest$ </concr>)}

// STOP

forall $C_{i,j}$ generate a rule skeleton as follows:
  $S \leftarrow S \cup$ {<abst> _ </abst>
            (<concr> <k> $C_{i,j}$ => . </k> $c_i$ </concr>)}

// RESUME

forall $C_{i,j}$ generate a rule skeleton as follows:
  $S \leftarrow S \cup$ {<abst> <k> $busy$ => . ...</k> </abst>
            (<concr> <k> $C_{i,j}$ </k> $c_i$ </concr>) => .}
```

Figure 3.2: Meta-algorithm for SPLIT + WRAP to generate a rule schemata

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.

Next we present a general, modular system as an instance of this modeling methodology, system that consists of both software and hardware components. Then, we discuss, at a general level in this chapter and in details in subsequent chapters (6 and 7), how to embed, both flow- and data-oriented, abstract executions (a la abstract interpretation and not only).

3.2 WCET Analyzer - Instance of the Modular System

Modern processors feature aggressive optimizations that influence the execution of programs. The WCET estimation in the presence of micro-architecture becomes harder, as micro-architecture elements introduce difficult to predict or even non-deterministic behaviors. Consequently, the instruction and data caches, as well as in-order pipelines modelings have been the most popular [41, 70, 74, 75] in the context of timing analysis. We propose in this thesis, a modular design for cache memories and a simple main memory model. Since modularity is our modeling target, we expect to be able to plug-in various micro-architecture elements (or variants of the existing ones), without changing the programming language definition.

Our design, in Fig. 3.3, relies on a number of modules, corresponding to the processor (language), the cache memories, and the main memory, all of which communicate using predefined message names. With respect to the previously introduced classification, our design consists of the following types of modules:

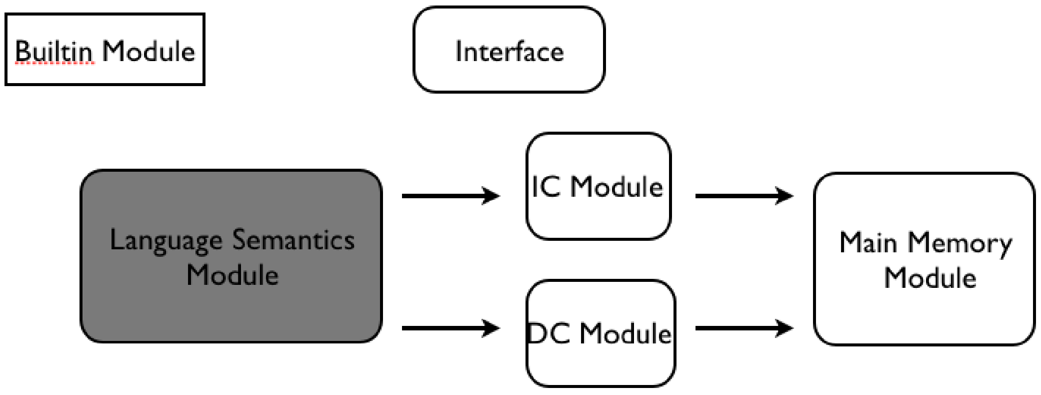


Figure 3.3: WCET Analyzer - The Module System

- A *support* module, module that ensures support operations for the entire $\mathbb{K}$ definition
- A *structural* module called *Interface* (in Fig. 3.3), which holds a sub-configuration with generic information (i.e. profiling, debugging, architecture parameters etc) and special operations to accommodate both concrete and abstract executions of programs over specified architectures.
- Four *functional* modules : the assembly language definition (M_1), the main memory system (M_2), the instruction cache module (M_3) and the data cache module (M_4). We specify that these are the only modules required for the concrete execution of a program in the presence of a specified architecture.

Our system for WCET analysis is built around the definition of the semantics of the *SSRISC* assembly language, presented in details in Chapter 4. To justify an important design decision with respect to the aforementioned modules' representation, we anticipate that the concrete configuration of the programming language would omit a store or memory cell. Such a semantic entity is indispensable to capture program executions, while running the semantics.

Therefore, to continue with our overview on how the current system for timing analysis is proposed, we decide to design the language semantics rules to update only the registers. In this way, the representation of the memory system is disconnected from register updates and therefore, amenable to refinements. One such refinement is the presence of the *IC Module* and *DC Module* for the instruction cache and respectively for the data cache. We emulate the organization of an assembly file into data and code sections, this leading to one representation for the *Main Memory Module* (module M_2), in Fig. 3.3. The sub-configuration c_2 for the functional module M_2 is presented next.

$$\mathit{MemConfig} \equiv \langle K \rangle_{\mathsf{k}} \, \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Instr}] \rangle_{\mathsf{cmem}} \, \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Data}] \rangle_{\mathsf{dmem}}$$

Now, similar sub-configurations with component-specific elements (i.e. replacement policies, ages, integer and floating point registers etc) are present in all other modules. Apart from the corresponding sub-configuration, the $\mathbb{K}$ module are compiled, using K-Maude - a $\mathbb{K}$ framework implementation, into Maude modules, which correspond to rewrite logic theories. While we refer to the definition for *Main Memory Module* in more details in Chapter 4, next we discuss another design issue.

In this proposed modeling methodology, an important part is the communication between modules. In the meta-algorithm, two functional modules M_i and M_j communicate using a set of messages $C_{i,j}$. With respect to our example, *Main Memory Module* receives requests for instructions `geti` and data `getd` from *Language Semantics Module*. Such messages form the *input interface* of *Main Memory Module*.

For example, the k cell processes the requests for instruction or data from the cache memories or the processor. In our initial design, *Language Semantics Module* issues the instruction requests using the `geti`(PC) operation. The memory system interprets the PC value as an address and checks this location in the code memory part, the cmem cell. There are two possible cases, covered by the two $\mathbb{K}$ rules in Figure 3.4. If the instruction is found in the code memory cmem, then `geti`(PC) rewrites to the actual instruction and the control is back to the processor - rule [MEM-INS]. If the instruction is not found

```
rule [MEM-INS] :
        <k> geti(PC) => incPC(PC) ~> Ins ...</k>
        <cmem>... PC:#Int32 |-> Ins:Instr ...</cmem>

rule [MEM-LAST] :
        <k> geti(PC:#Int32) => last </k>
        <cmem> CMem:Map </cmem>
when notIn(CMem, PC)
```

Figure 3.4: 𝕂-rules for an instruction request from the main memory

in the code memory `cmem`, a special token denoted as `last`, signals the termination of the execution - rule [MEM-LAST]. We rely on a special built-in function `notIn` to check if the instruction exists in the memory. Both `last` term and the returned instruction or data are part of the *output interface* of *Main Memory Module*. Similar input-output interfaces for all the functional modules in the system.

The 𝕂 framework enables the modular and executable semantics specification of programming languages. As mentioned previously, the key ingredient to system specification using 𝕂 is through configuration manipulation. *Main Memory Module* is parametric, as it is the formal executable semantics of the assembly language (through the support *Builtin Module*) and the specified behavior of instruction and data cache memories. In the case of *IC Module* and *DC Module*, this parameterization refers to the cache parameters (size, associativity), the replacement policies (Least Recently Used (LRU) and First-In First-Out (FIFO)) and, for the data cache alone, the writing policies (write-through no-allocate and write-back with-allocate).

The execution of a given program (in assembly language) over a specified underlying micro-architecture involves the four functional modules - corresponding to the assembly language semantics, the two types of cache memories and the main memory. The actual computation takes place in *Language Semantics Module*, while the others provide, via communication, the necessary information. This design relies on a sequential computational style: a message that is issued by one module is received by exactly another module. With respect to the kind of data that is propagated during a computation, there are concrete executions and abstract executions. While the latter category requires extensions of the module system in Fig. 3.3, the former one is handled by it, starting with a ground initial state.

For example, if the current execution step requires a particular load instruction, the concrete computation starts with a fetch request for this instruction, using the `geti` message between *Language Semantics Module* and *Main Memory Module* (assume the system without cache memories). When the instruction is returned, the language semantics issues an operand with a `getd` message, and upon receiving it, the actual load operation takes place. This kind of interleaving between communication requests and data is exploited, in a novel way, to inject abstract behaviors over this concrete system.

3.3 Interface Module

The system is designed to simulate the execution of a program on a specified, underlying architecture, and the $\mathbb{K}$ module corresponding to the language semantics plays the role of the processor. The key element of this design is how the computational cell `k` is used. The modules of the concrete system share the same `k`, with the communication being done via this cell. A special module keeps all these communication messages and facilitates, in this way, further structural and functional extensions. Next, we elaborate on the content and the usability of this module.

The $Interface\,Module$ is a structural module and has a sub-configuration with shared $\mathbb{K}$ cells (i.e. for profiling, debugging, architecture parameters etc) among functional modules.

$$ItfConfig \equiv \langle Message \rangle_{\mathsf{debug}} \, \langle \mathsf{Map}[Name \mapsto Value] \rangle_{\mathsf{profile}}$$

$$\langle \mathsf{Map}[Name \mapsto Value] \rangle_{\mathsf{params}} \, \langle \mathsf{Map}[IName \mapsto Value] \rangle_{\mathsf{tmodel}}$$

The debug cell contains special information of sort String - *Message* - to trace specific $\mathbb{K}$ rules (used mostly with the underlying Maude's tracing facilities). The mapping in profile records, currently, information about instruction and data cache hits and misses. Its storing capability could be easily extended to record pipeline conflicts, stalls or other micro-architecture behavior with impact on timing analysis. A similar functionality with profile displays the cell params, where the cache related organizational aspects (i.e. sizes, associativity) and functional aspects (i.e. replacement policy type, writing policy type) are specified. *Name* and *Value* stand for the attribute's name (i.e. hiti for cache instruction hit) and respectively its pre-assigned value.

Our definition and use of the tmodel cell emphasizes an important aspect of the modular design - it accommodates various timing models, without changing the actual formal executable semantics, nor the micro-architecture descriptions. We understand a timing model as a valued representation of atomic behaviors in the specified system. The definition of atomic is loose, we consider the timing model at the level of instructions (with respect to the formal executable semantics) and at the level of cache memory interactions (with respect to the micro-architecture behavior).

It also contains the declarations for the communication messages and auxiliary parameters. For example, a list of partial message names and signatures, annotated with the corresponding strictness attributes are presented in Fig.3.5. The message geti is a request for an instruction fetch, the messages getd and getdDbl are requests for word and double-word data (corresponding to the memory load operations), the messages putd and putdDbl are requests to write word and double-word data (corresponding to the memory store operations).

From the *Main Memory Module* perspective, the entire set of messages from Fig.3.5 form its input communication interface, while the requested instruction or data, or a successful write acknowledgement (for the last two messages) represents its output interface.

```
syntax Messages ::= geti ( K )
                  | getd ( K )
                  | getdDbl ( K )
                  | putd ( K , K ) [strict(2)]
                  | putdDbl ( K , K , K , K ) [strict(2 4)]
```

Figure 3.5: Declarations for the message names (partial)

3.4 The Builtin Module

Our system design consists of several important modules, shown in Fig.3.6. These are: a builtin module *builtins* and its extensions to symbolic values (for constant propagation and interval analysis) and operations *s-builtins*, a "glue" module, *ssrisc-settings* to specify how the language semantics, represented by *ssrisc-lang* and the main memory, *ssrisc-mem*, communicates.

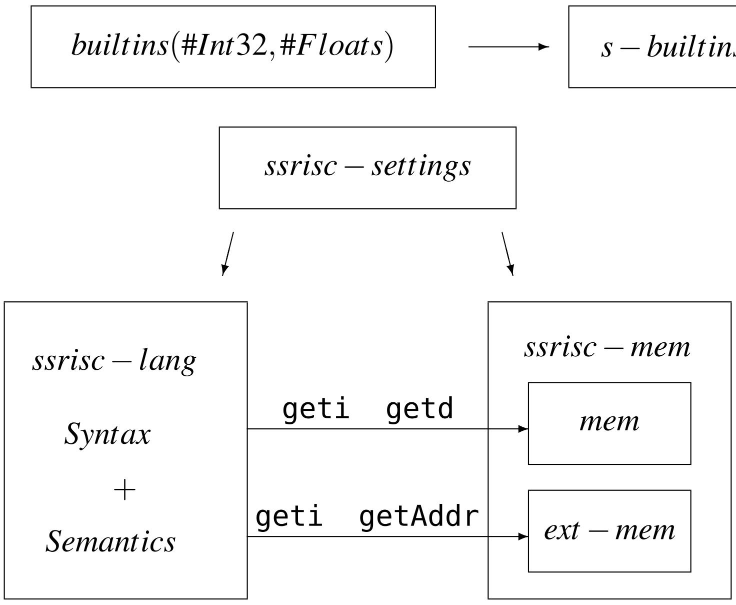


Figure 3.6: The system organization: $ssrisc - settings$ keeps the communication channels between the language definition $ssrisc - lang$ and various main memory models mem and $ext - mem$

The language specifics influence the language design in the following way: the semantic rules of the assembly language instructions use various representations for integers (signed and unsigned, on 32-bit, 16-bit or 8-bit format) and floating point numbers (single and double precision). All these, together with constant declarations, arithmetic and logic operations, conversion operations between these types and auxiliary checks (i.e. overflow, comparison with zero) are in the special *Builtin Module*, developed on top of the standard programming language builtin provided by $\mathbb{K}$. For example, the signed addition between two integers, `+Int32` is implemented over the `+Int` operation. The current implementation uses the predefined Maude number representations for both integers and floating points.

This *Builtin Module* is particularly important in the context of defining abstract executions. In the subsequent chapters, we present several extensions for this module. We define a symbolic version of it, when the support, concrete operations are overloaded to manipulate symbolic numerical values. For example, an arithmetic addition `+Int32` between two operands of sort $\#Int32$ is able to handle operands of sort $\#SInt32$, an extension of $\#Int32$. Applicability of such an extension is illustrated in Chapter 8, when we discuss a data analysis to detect a particular kind of infeasible paths. Moreover, the *Builtin Module* is further extended to accommodate another important abstract domain - the intervals. For example, the same arithmetic addition `+Int32` is now extended to handle operands of the sort $\#IInt32$, represented as intervals.

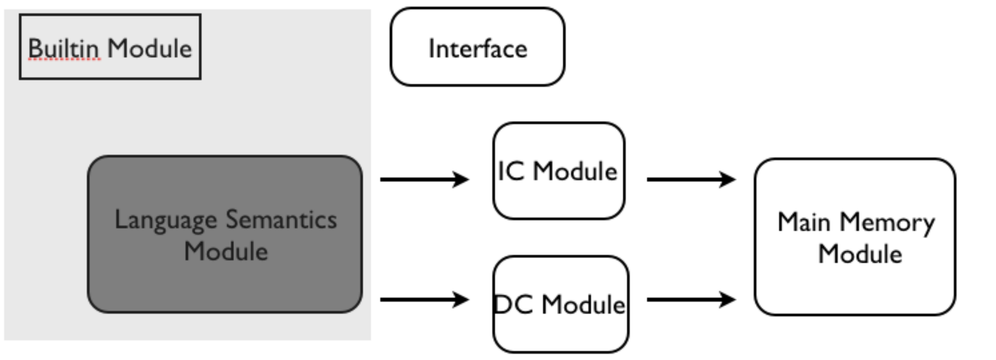
Both the formal executable semantics and the micro-architecture specification are designed to use the *Builtin Module* for the language operations. Since this module enjoys both concrete and symbolic domains, the $\mathbb{K}$ definition of the entire system is parametric with respect to the support operations, simplifying the process of abstraction definition. The chapters 7 and 8 treat in detail this topic.

3.5 Modules for Abstractions

The modularity of our design allows us not only to extend the concrete system with new functionalities, expressed as new modules, but to define abstractions directly over the existing modules, which are kept unmodified. While, these modules are not directly represented in the general module system, in Fig.3.3, we prefer a notational convention, using shaded areas that cover all the necessary modules to define abstractions of interest. There are several such examples, as follows:

- In Fig.3.7 (top) the shaded area, which covers *Builtin Module* and *Language Semantics Module*, represents the abstract wrapping for an interval analysis on an assembly language program.
- In Fig.3.7 (bottom) the shaded area, which covers *Language Semantics Module* and *Main Memory Module*, represents an abstract wrapping to extract ILP structural constraints from the input program
- In Fig.3.8 the shaded area, which covers *Language Semantics Module*, *IC Module* and *Main Memory Module*, represents an abstract interpretation-based analysis to classify the input program's instructions with respect to their cache memory behavior.

For all abstract modules, the corresponding abstract configuration has two components: (a) a definition-based set of semantic entities, derived from the concrete definition of the system and (b) an abstraction-specific set, to represent collections of abstract data and auxiliary elements necessary to compute the abstract result. If we refer to the second point above, the abstraction for the ILP structural constraints generation uses as (a) a wrapped configuration over the language definition (with the computation cell k and the program counter pc cell) and the main memory module (with the code memory component of the mem cell).



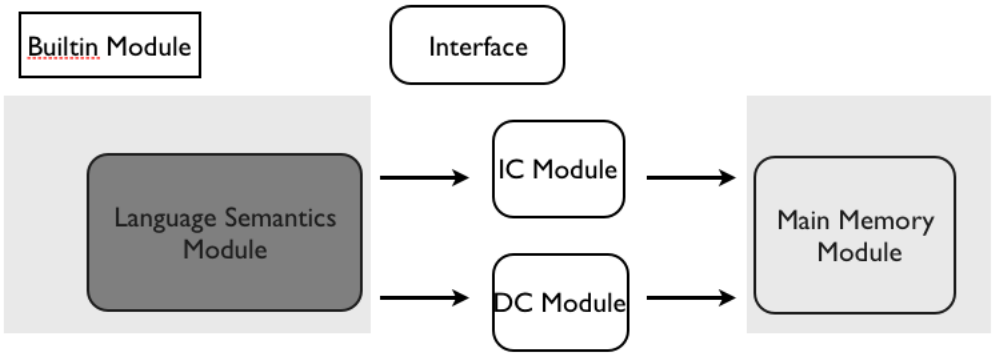


Figure 3.7: Module system for interval analysis (top) and ILP structural constraints extraction (bottom) - represented with shade area

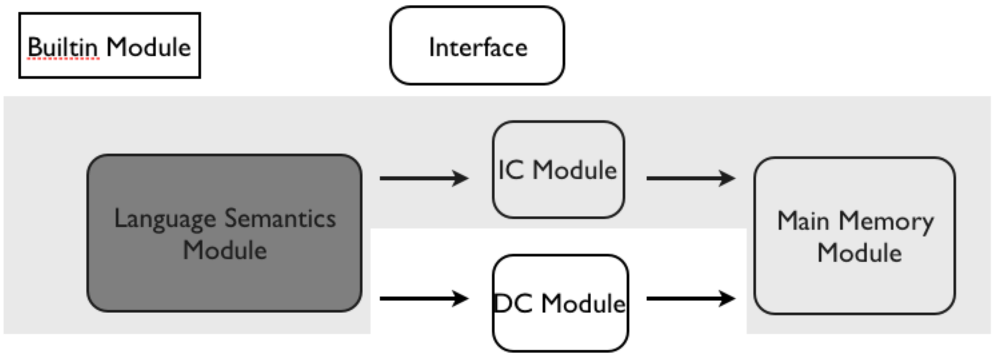


Figure 3.8: Module system for AI-based instruction cache behavior - represented with shade area

Similarly, for the third point above, the AI analysis for the instruction cache behavior computes sets of abstract cache states and therefore requires a wrapping of the instruction cache module, with the cache content represented by the ic cell, the ages and the replacement policy, represented by ages and repl $\mathbb{K}$ cells.

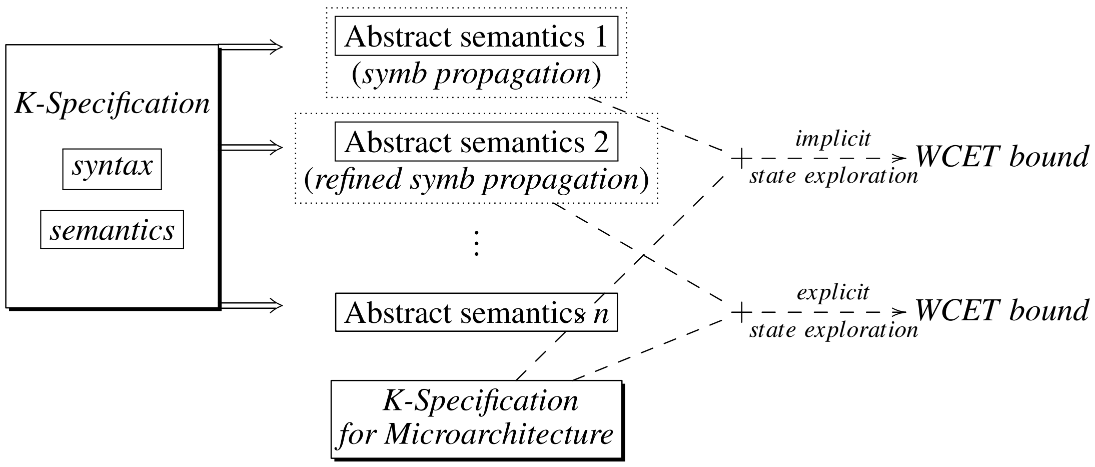


Figure 3.9: General methodology for WCET estimation, based on the 𝕂 framework

The execution of a program over a wrapped configurations is called *the abstract execution*. In the Chapter 7 and 8, we present implementation specific details of such abstract executions, in particular of the ILP+AI methodology for WCET analysis and reachability analysis for control flow graph (CFG) extraction or path analyses. The two general program reasoning methodologies, using explicit or implicit state exploration are summarized in Fig.3.9.

3.6 Case Study: Design Issues in the 𝕂 Definition for the ILP + AI Methodology

Next, we address several important design issues, using as an example the standard methodology of the ILP + AI towards WCET analysis. Similar observations stand for the other path analyses (based on data abstractions), shown in Chapter 8.

Working with the 𝕂 framework brings the following view on developing program analysis tools: give the formal executable semantics of the language of interest and use it to develop abstractions. The 𝕂 framework provides a specialized notation to easily manipulate program configurations and transitions using rewriting techniques.

The 𝕂 framework-based encoding of the ILP+AI approach leads us to some points on how to define abstract executions for timing analyses. We follow the same general view: encoding abstractions for path and processor behavior analyses require modifications to the concrete state and/or the transition relations between these states. We follow these from the point of view of the design decisions.

1. *Filter the state entities.* The configuration of a formal semantics of a programming language contains all the necessary semantic entities to define and run concrete programs. This rich description serves as basis for tailoring particular abstract executions, by manipulating these semantic entities. The two analyses for ILP constraints extraction and for processor behavior (i.e. the instruction cache behavior) rely on the abstract configurations, for both ILP and AI parts. The *PC* value is central to both these analyses. The abstract execution to generate and collect the incoming and outgoing edge constraints for each program point, relies, of course, on the *PC* value. Similarly, the actual instruction cache activity is reduced to address calculation and contention among these addresses using the replacement policy.

2. *Wrap the abstraction.* The $\mathbb{K}$ notation allows to describe semantic transitions between states, using only the necessary pieces of configuration. Therefore, the transition from concrete to abstract definitions should be done with a limited number of changes, as well. The two abstract semantics of the ILP + AI approach would be encoded in the following general way: a meta algorithm coordinates the execution of the wrapped abstraction and the concrete counterpart (language semantics and the architecture definition). For example, the cache behavior abstraction uses the same replacement algorithm, that is already implemented in the concrete description. The only difference is that the algorithm is applied on sets of concrete states (an abstract state) and the replacement algorithm gets applied

several times during computation of one abstract state.

3. *Incremental development of abstractions.* The $\mathbb{K}$ framework's ability to manipulate configurations and transitions also offers the possibility to incrementally develop abstractions for a particular problem. This is inspired from rewriting logic, via equational abstraction [85] and from existing solutions to the WCET analysis [113]. For example, the abstraction for ILP constraints extraction is driven by values of the program counter *PC* and therefore, it requires manipulation of symbolic data, via a simple data abstraction. Another example, the proposed optimization to reduce the number of constraints, via on-the-fly basic blocks detection, extends the abstract configuration and the corresponding $\mathbb{K}$ rewrite rules.

Next, we survey our viewpoint on how the $\mathbb{K}$ framework helps to solve some of the issues arising when we aim for the modularity and the incremental development of abstractions. We refer in advance to the $\mathbb{K}$ definition of the ILP + AI methodology, presented, in details, in the subsequent chapters (7 and 8).

1. *Separation of Concerns.* The $\mathbb{K}$ framework, through the specialized notation, describe the rewrite rules in a concise manner. This leads to an automatic separation of concerns when we define the abstract semantics. For example, in the case of the abstract execution for the ILP constraints extraction, the point of interest is to generate, for a program point, flow information as incoming and outgoing edges.

   Since the information regarding incoming and outgoing edges is not explicitly represented in the concrete semantics, the [FETCH] rewrite rule, in Fig. 8.1, connects the abstract rules with the cell that holds the program. Also, a $\mathbb{K}$ definition allows language operations to produce useful data in the cells. For example, both the concrete instruction cache behavior and the abstract execution for the instruction classification use the helping function for the concrete placement computation, directly (in concrete) in rule [PLACE1] in Fig. 5.5 or indirectly (in abstract) via the abstract cache replacement policy `replace`, in all the rules in Fig. 8.6.

2. *Implementation Reuse.* The two abstractions are guided by an algorithm that transfer the execution to the concrete counterpart whenever it is possible. For example, the operations of cache update and cache replacement could be reused, with minimal modifications, in the abstract execution for instruction cache behavior. For presentation purposes, we opt to encode the cache update directly in the rewrite rules [HIT] and [PLACE2], in Fig. 5.5 and the cache replacement, via a support operation called `compPl`, in rule [PLACE1]. As expected, the instruction cache abstraction in Fig. 8.6 does not reuse the cache update, as seen from the way the aic cell is modified, but reuses the replacement algorithm, wrapped up in the `replace` operation, in all the rewrite rules.

Our modular system is designed to facilitate smooth abstraction embedding, without changing a core system. We target a tool for WCET analysis, based on cooperation on various abstractions, with a particular combined method of ILP + AI standing out. The most successful implementation of this methodology for WCET analysis is the tool `aiT` by AbsInt [1], used in the aeronautics and automotive industries. Our work investigates an encoding of this combined method, from a slightly different point of view. We start with the $\mathbb{K}$ definitions of the assembly language and the underlying micro-architecture and explore if/what pieces of concrete behavior could be reused in the abstract executions.

3.7 Related Work

We propose a generic modular system, where the constitute modules communicate through a message passing mechanism, using the underlying $\mathbb{K}$ framework technology.

The k cell plays the role of a special data bus, with information being put on by both the processor, a role played by the *Language Semantics Module* and the other components of the system.

The work in [51] presents 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. While we address a similar architecture there are 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 [51] 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 the *Builtin Module*. 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 about 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 *Language Semantics Module* from the data and code memory, in the

Main Memory Module.

The $\mathbb{K}$ framework has a collection of language studies, among them ithe complex benchmark language called Challenge and used as a case study in [101] and the language Agent, described in [29]. Both of these languages use multi-paradigm concepts and because of this, they are implemented in a modular fashion. For example, the Agent language supports if and while statements, references, threads, code generation as quote-unquote mechanism, special control manipulating statements as halt and call with continuation etc. Each of these are implemented in a separate $\mathbb{K}$ module, with the underlying k cell is available in each. In our case, k plays a more elaborate role, as the modules communication through k via communication messages,

Chapter 4

$\mathbb{K}$ Definition of *SSRISC*

In this chapter we present the formal $\mathbb{K}$ definition for a RISC assembly language, part of the MIPS family of assembly languages and supported by the Simplescalar toolset [18]. It is the *Language Semantics Module* in the general module schema presented in the previous chapter. As we previously mentioned, we refer to this language as *SSRISC* [4].

To define a programming language in the $\mathbb{K}$ framework means to cover both syntactic and semantic aspects. The latter category relies on the definition of a programming language configuration, whose transformations are followed when the rewrite rules are defined, and on the state (or storage, or memory) part of the configuration.

We organize the presentation as follows: first we introduce the language syntax, then the language semantics, both using $\mathbb{K}$ code. Out of the three forms of $\mathbb{K}$ notations we opt for the ASCII one. Then, we introduce the specification for the main memory system, tailored on our main target application code - disassembled executables. Then, we present, as an extension (several are actually possible) of this language, projected at the level of the memory system as well. In the end, we relate our definition to the existing approaches with respect to the rewriting-style specification of systems, as well as other definitions.

4.1 Syntax

For presentation purposes, we select several representative instructions of *SSRISC* that emphasize on all aspects of the definition, from auxiliary operations to a general desiderate of constructing the rewrite rules - *one rule per instruction*.

The 𝕂 annotated syntax of a subset of the *SSRISC* assembly language is given in Figure 4.1. The left column shows the abstract syntax, in BNF form, while the right column states the corresponding 𝕂 strictness attributes that give the evaluation strategies of the declared operators. More precisely, the `strict` attribute tells that the enlisted operands are reduced to base values of (𝕂 builtin) sort *KResult*. The strictness attributes are actually syntactic sugars in 𝕂, compactly encoding a set of structural rules that achieve the same result as reduction via evaluation contexts.

For example, the `add` instruction is `strict` on the second and third operands, which implies that the last two registers, called sources, are reduced to values before the actual addition takes place and the first, destination register is updated with this value. When `strict` appears without arguments, like for `mult`, it means strict in all the operands. With respect to typing, registers *Reg*, immediate operands (as in the `addi` instruction) *Imm*, addresses *Addr* and address offsets *Off* are of sort *#Int32*. The floating-point subset of *SSRISC* is represented by instructions such as `add.s` and `div.s`.

This sorting mechanism permits our *SSRISC* language definition to execute the input programs with concrete values. As we mentioned previously, this definition is parametric in the sorting representation, using the supersorts *#SInt32* and *#IInt32* of *#Int32* it allows us to execute the programs with symbolic values, respectively with values represented as intervals. We elaborate on these extensions in the later chapters when we present reasoning-based techniques encoded over *SSRISC* assembly language definition.

We present a number of *SSRISC* instructions, grouped in arithmetic-logic instructions, branch and jump instructions, load and store instructions, special instructions. Moreover,

```
Instr ::= add Reg , Reg , Reg ;             [strict (2 3)]
          addi Reg , Reg , Imm ;              [strict (2)]
          mult Reg , Reg ;                        [strict]
          div Reg , Reg ;                         [strict]
          and Reg , Reg , Reg ;             [strict (2 3)]
          or Reg , Reg , Reg ;              [strict (2 3)]
          j Addr ;
          jr Reg ;                                [strict]
          jal Addr ;
          beq Reg , Reg , Addr ;            [strict (1 2)]
          bne Reg , Reg , Addr ;            [strict (1 2)]
          lw Reg , Off ( Reg ) ;              [strict (3)]
          lw Reg , ( Reg + Reg ) ;          [strict (2 3)]
          sw Reg , Off ( Reg ) ;              [strict (3)]
          sw Reg , ( Reg + Reg ) ;          [strict (2,3)]
          break ;
          nop ;
          bc1t Off ;
          l.s FReg , Off ( Reg ) ;            [strict (3)]
          s.s FReg , Off ( Reg ) ;            [strict (3)]
          add.s FReg , FReg , FReg ;        [strict (2 3)]
          div.s FReg , FReg , FReg ;        [strict (2 3)]
          ceqs FReg , FReg ;                      [strict]
```

Figure 4.1: The 𝕂 annotated SSRISC language syntax: BNF syntax of SSRISC instrutions on the left and their 𝕂 strictness attributes on the right.

the instructions work on integer operands, like the registers *Reg*, immediate values *Imm* or floating point values *FReg*. The set of instructions selected for presentation purposes, in Figure 4.1, are classified as follows:

- The arithmetic-logic (ALU) group includes signed addition with overflow with register-based `add, add.s` and immediate `addi` operands, instructions for logical and/or `and` and respectively `or`, multiplication `mult` and division `div, div.s`; the latter category poses the "divide–by-0" issue. Also, the *SSRISC* language's instructions can manipulate operands of various sizes and types, and therefore, there are conversion and test operations between these types of data (i.e. single to double precision). One such instruction, `ceqs`, is the test between two floating point, single precision, registers.

- The branch and jump group of instructions include the indirect branch instruction `jr Rs`, unconditional jumps `j` and `jal` (for function calls and returns), if-then-else instructions such as `bne` and `beq`. There is also a set of special control instructions, for example `bc1t` based on the value in a special register, called floating point condition code.
- The load and store instructions present displaced and indexed addressing variants, in case the target register is integer `lw Reg, Off(Reg)` and `lw Reg , (Reg + Reg)`, and similarly, for floating point register `l.s` and `s.s` (two variants each, only one is included in this subset).
- The special group of instructions include, among other, an error indicator instruction called `break` and the classical no-operation instruction `nop`.

4.2 Semantics

The key element when we define the formal semantics of *SSRISC* is to establish the language configuration. In general, the configuration in the $\mathbb{K}$ framework is a wrapped multiset of cells, written in $\mathbb{K}$ mathematical notation as $\langle cont \rangle_{\mathsf{lbl}}$ and in ASCII form as `<lbl> cont </lbl>`, where *cont* is the cell content (possibly itself a multiset of cells) and lbl is the cell label. The $\mathbb{K}$ cells hold the necessary semantic infrastructure (registers, instruction cache, memory, etc.). Two cells appear in most $\mathbb{K}$ definitions: a cell top whose label is $\top$ that encloses all the other cells, and a cell labeled k that holds the computation (syntax). For presentation purposes, we omit to include the top cell in the subsequent definitions of the language configurations.

The $\mathbb{K}$ configuration for the *SSRISC* language is:

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

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

The `k` cell maintains the current computation, i.e., the current program or fragment of program. The computations, i.e. terms of special sort K, are nested structures of computational tasks. Elements of such a list are separated by an associative operator $\curvearrowright$, as in $s_1 \curvearrowright s_2$, and are processed sequentially: s_2 is computed after s_1; the identity of $\curvearrowright$ is denoted by “$\cdot$”. This particular cell has another important usage - transmit/receive messages - in the context of the modular design that we discuss in Chapter 3.

Simplescalar PISA instruction set has a number of 67 registers, such as:

- `zero` - the zero-valued source register
- `gp` - the global pointer register
- `sp` - the stack pointer register
- `ra` - the return address register
- `fcc` - the floating point condition code
- `f0 - f31` - the floating point registers

We design our formal executable semantics with an emphasis on several special registers, assigning them specific $\mathbb{K}$ cells.

The language configuration has two maps of registers to store integer and floating point values. The `regs` cell contains a set of integer registers and is a mapping from register names *Reg* to stored values *Val* (which is of sort $\#Int32$). Out of the enumerated registers, we design the content of the `regs` cell to include the registers `zero`, `sp` and `gp`. Similarly, the `fregs` cell contains 32 floating point registers - `f0 - f31`.

```
main (void) {            |    .rdata              |
  int a[3]={-2,1,4};     | a:  .word -2           |
  int i, sum=0;          |    . . .               |   . . .
  for (i=0; i<3; i++)    |    .text               |
      sum=sum+a[i];      |    la $3,a             | 1000 lui $3,4096
                         |    lw $4,0($3)         | 1004 lw $4,0($3)
  return sum;            |    . . .               |
}                        | L1: lw $2,32($fp)      | 1028 lw $2,32($30)
                         |    slt $3,$2($3),3     | 1032 slti $3,$2,3
                         |    bne $3,$0,L2        | 1036 bne $3,$0,1044
                         |    j L3                | 1040 j 1060
                         | L2: . . .              | 1044 . . .
                         | L3: . . .              | 1060 . . .
```

Figure 4.2: C program (left) with snapshots of direct assembly code (middle) and disassembled code (right)

`lo` and `hi` are special registers, required by the `mult` and `div` instructions to hold parts of the computed results. Also, the special register `ra` stores the return address, after a function is called and executed. This particular register is used by a special jump instruction - `jal` - jump address and link. The `break` cell is used, in the strict sense, by the instruction `break` and, in the more general sense, to capture program errors such as overflow or division by zero.

The value which indicates the current executing instruction (i.e. the program counter) is represented in a 𝕂 cell called `pc`. We opt to represent it in a different cell than the other registers, because we think it improves the readability of the semantics, especially on conditional and unconditional jumps.

The program configuration requires, as well, a representation of the main memory, that holds both the program and the necessary data. We discuss this design decision (i.e. variants of the *Main Memory Module*) and its trade-offs, in the later parts of this chapter.

Next, we present the concrete formal executable semantics for the *SSRISC* assembly

```
rule [SEM-GETI] :
        <k> . => geti(PC) </k>
        <pc> PC:#SInt32 </pc>
```

Figure 4.3: Rule for computation initialization

language, having the following goal in mind, to ease the process of deriving useful abstractions over it. We introduce the 𝕂 rules by means of defining the semantic rules of *SSRISC*. 𝕂 rules generalize the usual rewriting rules, in the sense that the 𝕂 rule manipulates only parts of the rewrite term, in three different ways: read, write and don't care.

We capture the execution of each *SSRISC* instruction in a number of successive steps: instruction request for instruction cache or memory, data request from data cache or memory, actual processing and finally, machine state update. This view with respect to the program's execution, via message passing between modules, is covered in Chapter 3.

Our design target is to capture the language semantics in a correct and concise manner, and, as a result, we propose *a single rewrite rule per each SSRISC instruction*. To achieve this, we extract some common functionality, as general register lookup and update, or use some wrappers as in the case of `pc` register update, or conversion operations between various data types. Next, we present a group of 𝕂 functions and rewrite rules that correspond to the language operations:

We start with the integer register lookup and update operations, the rules [SEM-R-LOOKUP] and respectively [SEM-R-UPDATE]. There are two corresponding 𝕂 rewrite rules, for similar operations on the floating point registers. These operations require, for the integer registers, only the cells `k` and `regs` and for the floating point registers, the same `k` cell and `fregs`.

If the current computational task is a integer register lookup, for a register *R*, as shown in the rule [SEM-R-LOOKUP] in Figure 4.4, the resulting configuration has the corresponding value *I* of *R* from the register cell. This 𝕂 rewrite rule brings a

```
rule [SEM-R-LOOKUP] :
        <k> R:Reg => I ...</k>
        <regs>... R |-> I:#SInt32 ...</regs>

rule [SEM-R-UPDATE] :
        <k> updateReg(I1:#SInt32, Rd:Reg) => . </k>
        <regs>... Rd |-> (_ => I1) ...</regs>
```

Figure 4.4: Rules for integer register lookup and update

new mentioned element of the 𝕂 notation, the "don't care" part of a list or multiset term, represented with ellipses. We prefer to use the sort #*SInt*32, which anticipates the extension of the sort #*Int*32 to handle symbolic values. The rewriting in the rule [SEM-R-UPDATE] takes place in both `k` and `regs` cells. On top of the computation, the support function `updateReg` which takes as arguments a irreducible, basic, value I_1 and a register *Rd*, is consumed (i.e. reduced to $\cdot$). In the cell `regs`, the previously stored (and unimportant) value, represented by _, is rewritten to the value I_1.

The `pc` register update consists of the following three cases as shown in Figure 4.5. The first rule [SEM-R-INCPC] represents the automated incrementation before an instruction is fetched. The list of computational tasks has on top a special token, called `incPC`, which signals the default case of next instruction address. This particular token becomes an important communication message between the *Language Semantics Module* and the memory-related modules (i.e. *IC Module* and *Main Memory Module*). The second rule [SEM-R-PC1] addresses the case of a mandatory jump and updates the `pc` with a specified target address, V_1. The last rule [SEM-R-PC0] represents the fall-through case of a branch instruction and leaves the value of the `pc` register only with the previous normal incrementation. The last two rules allows to keep in check the desiderate of *one rule per instruction* definition of *SSRISC* language.

Arithmetic-logic instructions usually results in modifications of the register file (`regs` and `fregs`) or one special registers. Hence, all the 𝕂 rules for the arithmetic-logic (ALU)

```
rule [SEM-R-INCPC] :
        <k> incPC => . ...</k>
        <pc> PC:#SInt32 => PC +Int32 8 </pc>

rule [SEM-R-PC1] :
        <k> setPC(1, V1:#SInt32) => . ...</k>
        <pc> PC:#SInt32 => V1 </pc>

rule [SEM-R-PC0] :
        <k> setPC(0, _) => . ...</k>
```

Figure 4.5: Rules for the program counter register (increment and set)

instructions, in Figure 4.6 (for integer-based instructions) and Figure 4.7 (for floating-based instructions), transform the task in the `k` cell into a register update, using either the `updateReg/FReg` or `updateLo/Hi` support operations. The former takes two arguments, the register and the value to be written, whereas `updateLo` and `updateHi` require only the value, the name of the destination register is implicit. With respect to this, there are two special 𝕂 cells in the configuration, `lo` and `hi`.

The integer-based version of (signed) addition and division, the `add` and `div` instructions, as well as the floating-point division instruction `div.s`, have extra checks as they could lead to errors, from overflow and respectively division by zero. For example, the 𝕂 rule [SEM-R-ADD] states that the `add` instruction with the source registers having values V_1 and V_2 reduces to an overflow check, `ovf`, for the signed addition between these two values and, if necessary, followed by the destination register *Rd* update with the result. The two variants of the division instruction, `div` and `div.s`, when on top of the `k` cell, it reduces to a division by zero check for the denominator value, `div0` and respectively `divf0`, followed by the necessary register updates. The instruction `div` deposits the results in the `lo` and `hi` registers, while the instruction `div.s` in the destination floating-point register in `fregs`. The multiplication instruction `mult` also uses the two special registers to store the 64-bit result.

We discuss next the group of branch and jump instructions. These, shown in Fig-

```
rule [SEM-R-ADD] :
        <k> add Rd:Reg,V1:#SInt32,V2:#SInt32; =>
            ovf(V1, V2) ~> updateReg(V1 +Int32 V2, Rd) </k>

rule [SEM-R-ADDI] :
        <k> addi Rd:Reg,V1:#SInt32,V2:#SInt32; =>
       updateReg(V1 +Int32 V2, Rd) </k>

rule [SEM-R-MULT] :
        <k> mult V1:#SInt32,V2:#SInt32; =>
           updateLo((V1 *Int32 V2) \%Int32 (1 <<Int32 32)) ~>
           updateHi((V1 *Int32 V2) /Int32 (1 <<Int32 32)) </k>

rule [SEM-R-DIV] :
        <k> div V1:#SInt32,V2:#SInt32; => div0(V2) ~>
       updateLo(V1 /Int32 V2) ~> updateHi(V1 \%Int32 V2) </k>

rule [SEM-R-AND] :
        <k> and Rd:Reg,V1:#SInt32,V2:#SInt32; =>
          updateReg(V1 &Int32 V2, Rd) </k>

rule [SEM-R-OR] :
      <k> or Rd:Reg,V1:#SInt32,V2:#SInt32; =>
         updateReg(V1 |Int32 V2, Rd) </k>
```

Figure 4.6: Semantics rules for *SSRISC* ALU instructions (integer)

ure 4.8, transform the task in the `k` cell into a correct `pc` register update, which has the next instruction program counter, as a result of instruction fetch. All these 𝕂 rules use the `setPC` operation; with the first argument 1 it overwrites the value in the `pc`, and with 0 leaves it unchanged, as shown in Figure 4.5. We present three jump instructions: `j` jumps directly at a specified value `Addr`, `jr` jumps indirectly, at a value specified in a given register `Rs` and `jal` jumps to the address stored in a special register `ra`. This latter jump instruction is used for function calls, `ra` keeps the return address from a function call.

The [SEM-R-J] and [SEM-R-JR] rules for jump instructions change the program counter register with the values *Addr*, respectively the content of the *Rs* register. For

```
rule [SEM-R-ADDS]
       <k> add.s Fd:FReg,F1:#SgFloat,F2:#SgFloat; =>
        updateFReg(F1 +SgFloat F2, Fd) </k>

rule [SEM-R-DIVS]
        <k> div.s Fd:FReg,F1:#SgFloat,F2:#SgFloat; =>
        divf0(F2) ~> updateFReg(F1 /SgFloat F2, Fd) ...</k>
```

Figure 4.7: Semantics rules for *SSRISC* ALU instructions (floating-point)

the branch instructions, the rules [SEM-R-BEQ] and [SEM-R-BNE] cover both the "fall-through" and "taken" cases using a support function, called `Bool2IntSy`, for the function `setPC`. The support function `Bool2IntSy`, together with the implementations of `==BoolSy`, `=/=BoolSy`, `+Int32` etc are in the *Builtin Module* (and its extensions for symbolic or interval-based values).

A special branch instruction `bc1t` uses a flag register, called `fcc` as one of the operands. The comparison between this register's value and value 1 sets the new program counter value, as it is done for `bne` or `beq` instructions.

Figure 4.9 shows the 𝕂 rules for load and store, both for the integer and floating point registers. We explain first the load operations of *SSRISC*: two variants for the `lw` instruction and one for the `l.s` instruction. The 𝕂 rewrite rules [SEM-R-LW1] and [SEM-R-LW2] transform the load instruction into the destination register `Rd` update. The memory address is computed based on a register value V_1 and an offset *Off* or between two register values V_1 and V_2. The communication message `getd` takes this address to the destination memory module (either *DC Module* or *Main Memory Module*). Similarly for the `l.s` instruction, with the difference being the destination register which is from the floating-point register file, the `fregs` cell.

The 𝕂 rewrite rules [SEM-R-SW1], [SEM-R-SW2] and [SEM-R-SS] show how the `store` instructions `sw` and `s.s` are reduced to a memory write request using the communication message `putd`. In this case, the message has the memory address and the

```
rule [SEM-R-J] :
         <k> j Addr:#Int32; => setPC(1, Addr) </k>

rule [SEM-R-JR] :
        <k> jr Rs:Reg; => setPC(1, Rs) </k>

rule [SEM-R-JAL] :
      <k> jal Imm:#Int32; => setPC(1, Imm) </k>
      <pc> PC:#SInt32 </pc>
      <ra> _ => PC +Int32 8 </ra>

rule [SEM-R-BEQ] :
      <k> beq V1:#SInt32, V2:#SInt32, Addr; =>
    setPC(Bool2IntSy(V1 ==BoolSy V2), Addr) </k>

rule [SEM-R-BNE] :
      <k> bne V1:#SInt32, V2:#SInt32, Addr; =>
  setPC(Bool2IntSy(V1 =/=BoolSy V2), Addr) </k>

rule [SEM-R-BC1T] :
      <k> bc1t Off:#SInt32; =>
          setPC(Bool2IntSy(FC ==BoolSy 1), Off) </k>
      <fcc> FC:#SInt32 </fcc>
```

Figure 4.8: Semantics rules for *SSRISC* branch and jump instructions

source register *Rd* as arguments. The data cache and the main memory process the `getd` and `putd` requests (or in other words, the corresponding memory modules have these communication messages part of their input interface). We elaborate more on these messages in the next section, dedicated on the *Main Memory Module* and in the Chapter 5, on cache memories.

The last group of *SSRISC* instruction falls into the category of special instructions. For presentation purposes, we included only the `nop` and `break` instructions, with their respective 𝕂 rewrite rules in Figure 4.10. The rule [SEM-R-NOP] rewrites the instruction into the mandatory program counter increment, using `incPC`. The second rule, [SEM-R-BREAK] shows how in the `k` cell, the instruction rewrites into the `last` term that ends the computation. The special `break` cell updates to reflect termination after a program

```
rule [SEM-R-LW1] :
      <k> lw Rd:Reg, Off:#SInt32(V1:#SInt32); =>
  updateReg(getd(V1 +Int32 Off), Rd) </k>

rule [SEM-R-LW2] :
      <k> lw Rd:Reg, (V1:#SInt32 + V2:#SInt32); =>
  updateReg(getd(V1 +Int32 V2), Rd) </k>

rule [SEM-R-SW1] :
      <k> sw Rd:Reg, Off:#SInt32(V1:#SInt32); =>
  putd(V1 +Int32 Off, Rd) </k>

rule [SEM-R-SW2] :
      <k> sw Rd:Reg, (V1:#SInt32 + V2:#SInt32); =>
  putd(V1 +Int32 V2, Rd) </k>

rule [SEM-R-LS]
       <k> l.s Ft:FReg, Off:#SInt32(V1:#SInt32); =>
          updateFReg(getd(V1 +Int32 Off), Ft) </k>

rule [SEM-R-SS]
       <k> s.s Ft:FReg, Off:#SInt32(V1:#SInt32); =>
         putd(V1 +Int32 Off, Ft) </k>
```

Figure 4.9: Semantics rules for *SSRISC* load and store instructions

error. We mention that `last` is also used for normal termination of computation, when the execution reaches the next instruction after the last one in the program.

4.3 Main Memory System

One of the most important design issues is the separation between the computational capabilities and the memory/storage ones. In this section and in the next one we elaborate on various design possibilities of the *Main Memory Module*. In short, this module is functional and has an input and output communication interface. Since our main target is to utilize this *SSRISC* language definition for WCET analysis of programs, we consider the program's representation with the richest hardware-related information. Next, we

```
rule [SEM-R-NOP] :
      <k> nop; => incPC </k>
      <pc> PC:#SInt32 </pc>

rule [SEM-R-BREAK] :
        <k> break; => last ...</k>
        <break> _ => 1 </break>
```

Figure 4.10: Semantics rules for *SSRISC* nop and break instructions

elaborate on the disassembled executables produced by the Simplescalar toolset.

We emulate the organization of an assembly file into a code and data text, with the 𝕂 configuration for the main memory having, along with the k cell, the two corresponding cells, cmem and respectively dmem.

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

The k cell processes the requests for instruction or data that come from the cache memories or the processor. For example, the language semantics sends an instruction request to the main memory system, using the `geti`(PC) operation. The main memory system interprets the *PC* value as an address and checks this location in the code memory part, cmem cell. There are two possible cases, modeled with 𝕂 rules [MEM-R-GETI1] and [MEM-R-GETI2], and shown in Figure 4.11. If the instruction is found in the code memory cmem, and `geti`(PC) rewrites to the actual instruction, while the control is back to the processing module, *Language Semantics Module* or back to the instruction cache, *IC Module*. If the instruction is not found in the code memory cmem, the same special token we mentioned ealier, `last`, signals the termination of the program execution. We rely on a special built-in function notIn to check if the instruction exists in the code memory.

Before we proceed to explain the data requests from the dmem memory zone, we discuss the format of the input assembly file, in Figure 4.1 (right). Each instruction in

```
rule [MEM-R-GETI1]
      <k> geti(PC) => incPC ~> Ins ...</k>
     <cmem>... PC:#Int32 |-> Ins:Instr ...</cmem>

rule [MEM-R-GETI2]
     <k> geti(PC:#Int32) => last </k>
     <cmem> CMem:Map </cmem>
     when notIn(CMem, PC)
```

Figure 4.11: Semantics rules for instruction fetch request in the Main Memory Module

the program is preceded by the actual memory address where the instruction resides. We use this address as the program counter PC, each time incremented with 8, as it appears in the input file. Therefore, when we mention memory address of a particular instruction, we refer to the value of its PC.

The data memory requests are based on the memory address *Addr* wrapped using a special communication message getd. The set of rules is presented in Figure 4.12. As seen, there are several cases, based on the data type (in this case the data length) and on data's presence in the memory. The first 𝕂 rewrite rule [MEM-R-GETD1] is the obvious one - the data memory, in the cell dmem has the necessary data, Data, at the required address Addr.

The second rule [MEM-R-GETD2] corresponds to the case when the data in dmem and assumes it was not previously initialized or written. Hence, a new memory address Addr is added in the dmem cell, storing an initial value 0. An alternative is to initialize such a data with an unknown, symbolic value, which can be accommodated with changes at the builtin level. We refer to this issue in the implementation section.

The third rule [MEM-R-DBL1] has on top of the current computation list, a double-word data request. We rely on a special communication message, called getDbl to send the starting address for this double-sized data. This particular 𝕂 rewrite rules covers the case when the data is not in dmem and two new addresses Addr and Addr +Int32 8 would hold it. Both values are initialized to 0 and then returned in the k cell.

```
rule [MEM-R-GETD1] :
         <k> getd (Addr:#SInt32) => Data ...</k>
         <dmem>... Addr |-> Data:#SInt32 ...</dmem>

rule  [MEM-R-GETD2] :
         <k> getd (Addr:#SInt32) => 0 ...</k>
         <dmem> DMem:Map (.Map => Addr |-> 0) </dmem>
when notIn(DMem, Addr)

rule [MEM-R-DBL1] :
         <k> getdDbl (Addr:#SInt32) => 0 ~> 0 ...</k>
         <dmem> DMem:Map (.Map => Addr |-> 0)
                         (.Map => (Addr +Int32 8) |-> 0) </dmem>
when notIn(DMem, Addr) andBool notIn(DMem, Addr +Int32 8)

rule  [MEM-DBL2] :
         <k> getdDbl (Addr:#SInt32) => V1 ~> 0 ...</k>
         <dmem> DMem:Map (Addr |-> V1:#SInt32)
                         (.Map => (Addr +Int32 8) |-> 0) </dmem>
when notIn(DMem, Addr +Int32 8)
```

Figure 4.12: Semantics rules for data fetching from the main memory

The forth rule [MEM-R-DBL2] handles a special case of the third rule - when the data memory `dmem` contains only the first half of the requested data. We work under the same assumption as before and we initialize the remaining data with 0, at a new memory address `Addr +Int32 8`. Obviously, there are several other case, similar to this latter rule [MEM-R-DBL2], which are not shown in Figure 4.12.

4.4 Extended Main Memory System

We mention in Section 3.4 that our implementation of the formal executable semantics of *SSRISC* is parametric with respect to the *Builtin Module*. There is another important design aspect - the separation between the memory (or storage) component from the other semantic pieces that define the *SSRISC* language state. Hence, we have modules like *Main Memory Module* or *IC Module*. One reason for this is to refine the memory modeling at the level of accommodating different types of cache memories (i.e. multi-leveled, joined instruction and data caches etc) or buffers (i.e. TLB buffers, load/store buffers etc).

Another reason to opt for this modular implementation is to accommodate further refinements of the main memory system, with minimal modifications of the language semantics (or better, without modifications at all). When we propose this, we have in mind an alternative assembly programs, represented in a more traditional way, with labels standing for actual memory addresses (for both instructions and data in the program). One such example is in Fig. 4.2 (middle), and is also part of the Simplescalar toolset capabilities (indirectly, via the `gcc` compiler that it uses).

We introduce next the labeled representation of a memory addresses and how the $\mathbb{K}$ framework caters for easy semantic extensions, both at the level of configuration and rules. In Fig. 4.2 (middle) there is an assembly language snippet that uses labels for instruction addresses (i.e. L_1, L_2) and for data addresses, (i.e. a). For presentation purposes, we show several strengths of the $\mathbb{K}$ framework, when we describe how to incrementally extend the assembly language definition to accommodate a labeled representation of a program. To keep things in a simpler perspective, we focus only on the data labels and therefore, we assume that all the instruction labels are a priori assigned to actual memory addresses.

We start with the new, extended configuration for the *Main Memory Module*, one that requires a label representation for the data addresses. We extend the previous memory configuration $Config_{MM}$ with a new cell, called `dlabels` which holds a mapping between symbolic addresses `SymbAddr` and concrete values `Val`. The `dmem` cell contains generated values for memory addresses. The configuration is presented next:

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

```
Instr ::= la Reg , Off ( Reg ) ;        |   [strict (3)]
          lw Reg , ( Reg + Reg ) ;      | [strict (2 3)]
          sw Reg , Off ( Reg ) ;        |   [strict (3)]
```

Figure 4.13: The 𝕂 annotated SSRISC language syntax: BNF syntax of SSRISC instrutions on the left and their 𝕂 strictness attributes on the right.

$$\langle \mathsf{Map}[\mathit{SymbAddr} \mapsto \mathit{Val}] \rangle_{\mathsf{dlabels}}$$

To explain this extended main memory design, first we give some insights on how this kind of symbolic address looks from the perspective of the *SSRISC* language definition. The representation of data memory addresses with labels has several characteristics. The *SSRISC* language has a number of instructions that permits manipulation of such symbolic addresses. Fig. 4.13 presents three such instructions - a new instruction called `la` - load address and two previously introduced instructions - a load `lw` and a store `sw` instruction. The new instruction `la` loads a destination integer register with the data that resides at a symbolic address. The sort of this address is denoted by `SVar` and is subsorted to the builtin sort for identifiers `#Id`. Then, the load instruction `lw` could handle symbolic addresses, based on a specified offset. A similar reason stands for the address computation of the `sw` instruction.

The 𝕂 semantics rules for the three previously mentioned instructions are shown in Fig. 4.14. For example, the rewrite rule [SEM-R-LA] describes the semantics of an instruction that manipulates symbolic data addresses into an integer register *Rd* update, via a structural 𝕂 rule. The symbolic memory address *VA* is wrapped by a new term, `getAddr`, which acts as communication message between the semantic and the memory model. The other two rules [SEM-R-LWA] and [SEM-R-SWA] behave in the same way.

Now, if we compare, for example the rules [SEM-R-LWA] from Fig. 4.14 with [SEM-R-LW1] from Fig. 4.9, we notice that the only difference is in the actual data address computation. The two functions `getd` and `getAddr` define different communication messages, which are sent to the *Main Memory Module*. Therefore, as expected, to ac-

```
rule [SEM-R-LA]
     <k> la Rd:Reg, VA:SVar; =>
            updateReg(getAddr(VA), Rd) </k>

rule [SEM-R-LWA]
       <k> lw Rd:Reg, VA:SVar + Off:#SInt32; =>
            updateReg(getAddr(VA + Off), Rd) </k>

rule [SEM-R-SWA]
       <k> sw Rd:Reg, VA:SVar + Off:#SInt32; =>
         putd(getAddr(VA + Off), Rd) </k>
```

Figure 4.14: Semantics rules for the extended label-manipulating instructions

commodate different memory models trigger modifications at the level of communication interfaces. In our example, the input interface of the *Main Memory Module* (and in mirror, the output communication interface of *Language Semantics Module*) incorporates the `getAddr` message.

The *Main Memory Module* with the corresponding sub-configuration $Config_{ExtMM}$ share with the previous described memory model, the two 𝕂 rules - [MEM-R-GETI1] and [MEM-R-GETI2] (i.e. having `geti` the top of the `k` cell). The data request rules get extended with a special case of `getAddr` communication message.

In Fig. 4.15 there are two 𝕂 rewrite rules with `getAddr` on top of the computation. The message `getAddr` is transformed into a lookup in the list of labels, in rule [SEM-R-DLABELS], to identify the value *Val* associated with a particular label *VA*. When this particular message is the result of the execution of a `lw` instruction , the actual data address is translated, using the information stored in the cell `dlabels`. There are several possible extensions for the *Main Memory Module*. We rule out, by assumption, the labels for instruction addresses, but these can be handled in a similar way, using a specialized cell.

Next, we address possibilities of testing the formal executable semantics of the *SSRISC* language. There are two possibilities. First, it is the standard manual testing

```
rule [SEM-R-ADDR]
     <k> getAddr(VA:SV) => Val ...</k>
     <dlabels>... VA |-> Val:#SInt32 ...</dlabels>

rule [SEM-R-DLABELS]
       <k> getAddr(VA:SV + V1:#SInt32) => Val ...</k>
       <dlabels>... VA |-> V2:#SInt32 ...</dlabels>
       <dmem>... (V1 +Int32 V2:#SInt32) |-> Val:#SInt32 ...</dmem>
```

Figure 4.15: Semantics rules for the getAddr in the extended memory module

- manual in the sense of constructing ad-hoc initial states and run the programs on interest. Second, it is also possible, to some extent, to automatically test the programs - automatically from the perspective of the initial state constructions.

With the extended *Main Memory Module* with the sub-configuration $Config_{ExtMM}$ we present such a refinement. We use this to construct the initial state of a program and to facilitate testing the *SSRISC* semantics. The alternative memory definition, with the sub-configuration $Config_{MM}$ which we obtain from disassembling the executables, does not help with this aspect. For the example program in Fig.3.6 (middle), `dlabels` contains the information $a \mapsto 100$, and `dmem` contains $\{100 \mapsto -2, 104 \mapsto 1, 108 \mapsto 4\}$, in other words, the values of the array a - same figure (left). The symbolic address is transformed into concrete value, for example 100.

An important element in our design methodology is that we can reason at the level of communication capabilities, to allow easy extensions to the definition. For example, the 𝕂 rules for the load and store instructions use `geti` and `getd` operations to fetch instructions, respectively data, from the memory. These two, for example, are placed in the $Interface\ Module$ and in this way, the memory model could be replaced with a different one, without affecting the semantics. Another example is with respect to the labeled representation of the data memory addresses. The instruction `la` uses the communication message `getAddr` to access the information in the cell `dlabels` (indirectly, in the *Main Memory Module*). `getAddr` is also placed in the $Interface\ Module$

A particularity of the *SSRISC* definition is that it allows to under-specify the memory content using symbolic values for stored data. For example, if a load instruction (i.e. `lw`, `l.s` and `la` accesses a memory address with an unknown value, this value is retrieved and further propagated during the program execution. The set of specific support operations on bitwise representation is extended to handle the symbolic value. These features are used in the context of abstract execution of programs for timing analysis [7].

The Simplescalar toolset [18] is an architecture simulator and presents two sets of instructions: a MIPS-based assembly language, used to compile C programs into it and a set of simulator-specific instructions. We implement the former, together with a number of pseudo-instructions and some of the instructions specific to the program with labels. Each instruction in the language is implemented using exactly one 𝕂 rewrite rule. On top of this, there are a number of auxiliary operations (i.e. set the program counter, overflow check etc). There are also two implementations of the *Main Memory Module*, with and without data labels, and many other extensions are possible (with a more refined data memory organization).

There are several facts with respect to the automated testing capabilities of the *SSRISC* semantics. Besides of the possibility to manually construct the initial state (i.e. registers and data memory contents) of the program, we allow a limited amount of automated testing, subjective to certain assumptions. For example, we need to produce assembly code without library function calls (i.e. memcpy), which currently, we do not support. We consider simple C programs, with small arrays of integer and floating point values, to test various arithmetic and logic instructions, as well as load/store instructions and conditional and unconditional jumps. Also, the return value of a C program and its corresponding representation in the assembly program - the value in a particular register - should be compared and decide the if the test passes or fails. We reiterate that our intention is to present the *SSRISC* language definition as a basis to define abstractions for timing analysis of embedded programs. These topics are covered in Chapters 6 and 7.

For this purpose, we need the non-labeled representation of the *Main Memory Module* because it contains useful information about instruction and data placement.

4.5 Related Work

Developing formal methods for assembly code is important for a wide range of applications. In computer security, there are techniques for detection of malicious code behavior [20] and protection against virus infection using code obfuscation [77]. In compiler design, the current approaches follow the principles stated in [55], either to generate correct-by-construction assembly code [71] or assembly code extensions, such as a type system [90], to facilitate its validation. Another important application is the worst-case execution time (WCET) analysis of embedded code, when it is necessary to guarantee execution time bounds for a particular program. For a survey on timing analyses, we refer the reader to [113]. All the previously mentioned applications require, usually ad-hoc, implementations of assembly language semantics. Our approach is to provide a formal executable semantics, which should serve as a basis for developing analysis methods.

In the same class of formal executable semantics of assembly languages there are the theorem-proving based approaches. One of the first works uses the ACL2 [63] prover to certify microcode programs for the Motorola CAP DSP [15]. The processor is modeled at both the instruction set and the pipeline levels. Another similar formalization is of ARM6 in [42] and ARM7 in [43], both using HOL theorem prover [45]. All these assembly language encodings are in the context of formal verification of the underlying architecture, and embeds correctness criteria with respect to the properties of interest. Our proposed 𝕂 language definition focuses on a different aspect, it is designed to be more modular to accommodate various formats of the input program, without changing the semantics.

To the best of our knowledge, there are two approaches to model assembly languages using Maude system. The work in [51] proposes a first model of a simplified RISC assembly language, used in verification of various microprocessor elements. The method presented in [110] presents a limited subset of the x86 assembly language for malware behavior detection. Both of these approaches focus more on verification issues and less on the language definition ones. Language semantics definitions rely on the memory system specification, both at the structural and functional levels. With respect to the memory representation in formal language definitions using rewriting logic, our current work complements the work in [53], which proposes memory models for imperative and object-oriented languages.

Chapter 5

Micro-architecture Modeling in $\mathbb{K}$

The execution of the real-time and embedded programs is greatly influenced by the underlying micro-architecture elements presented in the modern processors. This view brings a resource-aware programming perspective over the field, with the resources such as time, power, memory being under scrutiny.

Our *SSRISC* assembly language is based on the standard MIPS architecture, described in [95]. The Simplescalar toolset for architecture simulation uses a modified `gcc` compiler to generate MIPS-based executables from C programs. In summary, the MIPS architecture has the following characteristics: 32-bit registers with three-address and register to register instructions, the memory operations use indexed addressing and aligned words in the main memory. The memory related operations incur long delays in a program execution time. To improve the performances, modern architectures use special storage places to temporarily hold small contents of the main memory which are, or are speculated to be, frequently used. These small and fast memories are called cache memories [107].

We organize this chapter on cache memory modeling in $\mathbb{K}$, in four sections. First, we overview both organizational and functional characteristics of the cache memories, with an emphasis on their impact on the execution time of the embedded software systems.

Since we consider split-caches, the second and the third sections present the 𝕂 models for the instruction caches and respectively data caches, in the context of WCET analysis. The last part of this chapter is dedicated to a survey of works on the cache memories modeling.

5.1 Cache Memories

The cache memories are fast buffers that stand between the central processing unit (CPU) and the main memory system. The process of checking and updating these cache memories is complex, consisting of a number of steps. We briefly overview the (partial) flow of operations in cache memories and then we select a desired level of abstraction to capture the necessary information for timing analysis.

The caching mechanism starts operating when the CPU issues a virtual address, which is used to detect if the required data/instruction is already in the cache memory. A special buffer called *translation lookaside buffer* (TLB) maps this virtual address to real address (this translation consists of other smaller computation steps), which in turn is passed to a comparator for the hit/miss test. The TLB buffer is updated with the replacement status. Also, a part of the virtual address is used as an index to select a place in the cache memory (again there are some more smaller computation steps). In case of a miss, signaled as a comparison mismatch, then the real address is send to the main memory system.

Next we survey several cache memories characteristics and present our design for the modules corresponding to instruction cache *IC Module* and data cache *DC Module*, by abstracting away components (i.e. the TLB buffer, the comparator). We follow the design alternatives presented in [107] and succinctly comment on how we approach them from the 𝕂 perspective of their implementation:

- The cache fetch algorithm decides if the instruction/data is fetched on demand or is fetched speculatively. Our model considers the former case, with the information being requested by the CPU via messages such as `geti` and `getd`. In our modular design, presented in Chapter 3, the CPU role is played by the formal executable semantics of the *SSRISC* language. We treat the on demand fetching as communication messages between modules

- The cache placement algorithm returns the location or the set of locations where a particular instruction/data should be stored in the cache memory. This algorithm relies on the cache organization (i.e. cache associativity). We implement the cache replacement algorithms using functions, which are called to populate specific cells in the 𝕂 rules of the cache specification.

- The line *size* defines the unit of information that is transferred between the cache memory and the main memory. Both our instruction and data cache designs considers the size of a cache line equal to the size of a memory line. In other words, when an instruction is requested through the `geti` message from *IC Module* and, for example, it is not in the instruction cache, the result returned by the *Main Memory Module* is placed in a cache line.

- The replacement algorithm addresses the full cache case, when some information should be evicted from the cache to make room for the new information. There are several replacement strategies such as least recently used (LRU), first in first out (FIFO) or preudo-LRU (PLRU). We present these with respect to abstractions a la abstract interpretation implemented for caches with such replacement policies. One of our design goal is to integrate replacement policies without changing the 𝕂 rewrite rules of the cache behavior. Hence, the replacement policies are represented with functions, as in the placement policies case.

- The main memory update algorithm maintains the coherence between the instructions/data in the cache and the main memory. This separates the design of the instruction cache from the one of the data cache, with respect to the cache miss scenarios. For example, an instruction cache behavior has a cache miss on reading an instruction, while a data cache behavior is more complex, with cache misses on both reading and writing data.

- The miss ratio is an important design element to measure the quality of a cache memory. It could be measured when starting with an empty cache, called cold-start, or with a full cache, called warm-start. While this aspect is orthogonal to the WCET analysis (which should be safe and tight regardless of the input state of the cache memory), our design is able to keep profiling information for the concrete execution of the 𝕂 specifications of the cache memories.

- Another important aspect in the cache memory design is the separation of caches for instructions from data of a program. This design issue is captured by our two modules *IC Module* and *DC Module*.

- A data request from the main memory should return the last updated value. The presence of a data cache memory requires special care to maintain coherent information with respect to the main memory. There are several solutions to the cache - main memory coherence relation, we adopt a parametric implementation of a writing policy. Currently we support both write-back and write-through memory updates. We elaborate more on these in Section 5.3.

- Two other parameters - the cache size and the cache bandwidth - directly affect the cache memory performance. We design our 𝕂 definition of cache memories to be parametric in cache sizes, and we adopt a zero-time policy for the bandwidth (i.e. the rate at which data can be transferred to and from the cache).

The other three cache design parameters, the user/supervisor cache, the multicache consistency and the virtual versus real addressing issue, are not covered by our 𝕂 specification of cache memories.

We model a split organization of the cache memory and we concentrate all these requirements into two kinds of features. First, a cache memory is characterized by a number of parameters, such as cache size, number of cache lines, cache associativity. Second, a cache memory exhibits a number of behaviors such as information management or replacement policies. The state information, described using 𝕂 configurations, for both the instruction and data caches (1) share the cache parameters and profiling information and (2) poses some differences when handle cache misses.

Next, we present several generalities about (1). The cache size is the total number of bytes that can be stored. The cache line size is the number of bytes that can be transferred to and from the memory in one step. The associativity describes the relation between cache lines and memory blocks. A memory block can reside anywhere in the cache, in a group of cache lines or in exactly one line. This leads to the standard terminology of fully associative caches, for the first case, N-associative caches for the second - with N the number of cache lines and the direct-mapped caches for the last case. We assume the particular case of a cache line and a memory block having the same size.

With respect to the point (2), the data cache miss on a write operation poses specific problems. For example, there are two possible policies to maintain coherent information between cache and main memory contents, write-through and write-back. The former implies that the main memory is updated on every write, while the latter keeps the modified data in the cache, until the eviction. Therefore, the state information contains a list of "dirty bits" to emphasize inconsistent data between the cache and the main memory.

Our main goal is to define, in 𝕂, instruction and data cache behaviors which could be easily extended to accommodate abstractions for timing analysis purposes. These abstractions should equally work for fully-associative or direct-mapped caches, for caches with LRU or FIFO replacement policies etc. Therefore, we aim to a parametric representation of the concrete cache behaviors, in both in terms of the organization (size, associativity) and functionality (replacement policies).

The key element in a replacement policy model is the age information associated to each cache line. For example, let us consider a 4-line fully associative cache with the following cache content: the memory blocks *a* to *d* reside in the cache lines 1 to 4, respectively, with the block *a* being the youngest and *d* the oldest. A replacement policy identifies the memory block that should be replaced by newly requested blocks. Here is an example of the classical Least Recently Used (LRU). If the memory block *e* is requested and is not found in the cache, upon its retrieval from the main memory, the following two actions happen. First, the memory block *e* replaces the cached block *d*, which is the oldest, and second, the set of ages is updated, with *e* being the youngest, *a* the second youngest and *c* the oldest. The FIFO replacement policy shares parts of the functionality with the LRU replacement policies. We discuss next how we encode in $\mathbb{K}$ such replacement policies, aiming to maximize the $\mathbb{K}$ core reusability.

The $\mathbb{K}$ configuration of both the instruction and data caches include a special cell called `ages`, which keeps a mapping between cache addresses and corresponding age values. The replacement algorithm updates the `ages` cell, as described by the following rule stub, where *A* is the current map of ages and *cache_params* ensures the parametric implementation in terms of cache size and associativity

`<ages>` A `=>` $updAges(A, cache\_params)$ `</ages>`

We implement the replacement policies by rewriting rules, each $\mathbb{K}$ rule being annotated with the flag `anywhere structural`. The top computation task, represented by the `updAges` function relies on a sequence of four simpler sub-tasks to achieve the final age update that comes with the replacement policy. We present in Fig. 5.1 these functions with their corresponding signatures.

```
syntax Map ::=
     updAges ( Map , #Int32 , #Int32 , #Int32 , #Int32 )
   | updAgesList ( Map , #Int32 , List , #Int32 , #Int32 )
   | updAgesYoung ( Map , #Int32 , List , #Int32 , #Int32 )
   | updAgesOld ( Map , List , #Int32 , #Int32 , #Int32 )
   | updAgesFin ( Map , #Int32 )
```

Figure 5.1: Declarations for the modular implementation of age update algorithms

Since these special rewrite rules are shared by the instruction and data caches specifications, they are placed in the *Interface Module*, together with the communication messages. We opt for this apparently unfitting inclusion of the common functionality of the cache behavior specifications in the communication interface, to facilitate extending our WCET analyzer. In this way, all the design decisions related to cache parameters (i.e. part of the cache configuration, common functions, communication messages) are in the *Interface Module*.

The four 𝕂 rewrite rules in Fig. 5.2 implement the LRU replacement algorithm in a sequence of steps, the first two rewrite rules are generic with respect to particular *PID* replacement policy identifier and the last two are specific to the LRU policy.

The rule [ITF-UPDAGES] takes as arguments: the map *C*, the cache address *Pos* where the new information should be placed (and the corresponding age is updated as the youngest) and the cache parameters (cache size *NL* and cache associativity *M*). The right-hand side of the 𝕂 rewrite rule shows an auxiliary function `updAgesList`, whose purpose is to slice through the cache memory, to identify where the replacement should take place. The `getCLine` function is paired with the `gotCLine` from the second rule [ITF-UPDAUX] and returns a list *L* of cache lines, computed using the cache parameters *NL* and *M*. Once this cache slice is identified, then its corresponding age information is updated, via the `updAgesYoung` function.

The last two rules [ITF-UPD-YOUNG1] and [ITF-UPD-YOUNG2] present the particular implementation of the LRU replacement policy, identified with *PID* having the value

```
rule [ITF-UPDAGES] :
updAges( C:Map, wia(Pos:#Int32), NL:#Int32, M:#Int32, PID:#Int32 )
=>  updAgesList( C, wia(Pos), getCLine(wia(Pos), NL, M), M, PID )
[anywhere structural]

rule [ITF-UPDAUX] :
updAgesList( C:Map, wia(Pos:#Int32), gotCLine(L:List),
 M:#Int32, PID:#Int32 ) =>
    updAgesYoung( C, wia(Pos), L, M, PID )
[anywhere structural]

rule [ITF-UPD-YOUNG1] :
updAgesYoung( (Pos |-> 1) C:Map, wia(Pos:#Int32), _, M:#Int32, 0 )
=> updAgesFin( (Pos |-> 1) C:Map, 0)
[anywhere structural]

rule [ITF-UPD-YOUNG2] :
updAgesYoung( (Pos |-> Age:#Int32) C:Map, wia(Pos:#Int32),
   L1:List ListItem(Pos:#Int32) L2:List, M:#Int32, 0 ) =>
      updAgesOld( (Pos |-> 1) C, L1 L2, M, Age, 0 )
when (Age >Int32 1)
[anywhere structural]
```

Figure 5.2: Rewrite rules for updating the `ages` cell - the LRU policy (partial implementation)

zero. The newly added memory block, at the cache address *Pos*, becomes the youngest in the cache (i.e. represented with the value 1), in rule [ITF-UPD-YOUNG1]. All the other ages, corresponding to the cache addresses in the list *L*, gets updated recursively, in rule [ITF-UPD-YOUNG2], using the function `updAgesOld` (not shown here). The last function `updAgesFin` is used when the age updating terminates.

The implementation of the FIFO policy is similar, with the last two rules in Fig. 5.2 implementing the specifics of this replacement policy. These two policies LRU and FIFO play an important role in the degree of reusability that we attempt to achieve when we define abstract executions over this concrete specification of the micro-architecture.

5.2 Instruction Caches

The *IC Module* is designed to communicate with the *Language Semantics Module* and the *Main Memory Module*, a design decision that is reflected that at the level of input and output communication interfaces of these modules. Therefore, we refine the initial design, (a) in Fig. 5.3, to accommodate an instruction cache specification and a data cache specification in (b) and respectively (c) in Fig. 5.3. In this way, the main memory receives a request only in the case of a cache miss, either it is about an instruction or a program data. The two rules in Fig.3.4 are kept the same, except the request message name changes to `imiss`(PC) to reflect the instruction cache presence. The instruction cache configuration also uses a `k` cell to forward instruction requests from the processor to the main memory.

There are several instruction cache specifications of interest, with varying degrees of complexity. We present next two such instances, a stub implementation with a special role and a more accurate description which mitigates between its concrete representation and an abstract one, with respect to timing analysis.

The stub instance of the *IC Module* could be used to execute programs, forwarding the instruction fetch request to the *Main Memory Module* In this view, the instruction fetch request from the language semantics is forwarded, without any action, to the *Main Memory Module* and similarly with the return message. We recall that the *Language Semantics Module* sends a `geti` message in the `k` cell and expects an program counter incrementation followed by the requested instruction. This stub module has a correspondence in the unit testing methodology, where the same design strategy allows the user to isolate a desired behavior (specified in particular modules of interest). The inclusion of a stub module to replace an existing and more concrete implementation of it should be handled by an external tool.

Next, we describe the concrete instruction cache behavior. The configuration of the

$$\begin{tikzcd}
\text{(a)}\ \mathit{Lang\,Config.} \arrow[r, shift left, "\texttt{geti}(\mathit{PC})"] & \mathit{Mem\,Config.} \arrow[l, shift left, "\texttt{incPC} \curvearrowright \mathit{Instr}"]
\end{tikzcd}$$

$$\begin{tikzcd}
\text{(b)}\ \mathit{Lang\,Config.} \arrow[r, shift left, "\texttt{geti}(\mathit{PC})"] & \mathit{IC\,Config.} \arrow[l, shift left, "\texttt{incPC} \curvearrowright \mathit{Instr}"] \arrow[r, shift left, "\texttt{imiss}(\mathit{PC})"] & \mathit{Mem\,Config.} \arrow[l, shift left, "\texttt{iret}(\mathit{PC})"]
\end{tikzcd}$$

$$\begin{tikzcd}
\text{(c)}\ \mathit{Lang\,Config.} \arrow[r, shift left, "\texttt{getd}(\mathit{Addr})"] & \mathit{DC\,Config.} \arrow[l, shift left, "\texttt{Data}"] \arrow[r, shift left, "\texttt{dmiss}(\mathit{Addr})"] & \mathit{Mem\,Config.} \arrow[l, shift left, "\texttt{dret}(\mathit{Addr},\mathit{Data})"]
\end{tikzcd}$$

Figure 5.3: Communication types between the system modules

instruction cache is:

$$\mathit{ICConfig} \equiv \langle K \rangle_{\mathsf{k}}\, \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Instr}] \rangle_{\mathsf{ic}}\, \langle \mathsf{Map}[\mathit{ICParam} \mapsto \mathit{Val}] \rangle_{\mathsf{param}}$$

$$\langle \mathit{Addr} \rangle_{\mathsf{repl}}\, \langle \mathsf{Map}[\mathit{Addr} \mapsto \mathit{Val}] \rangle_{\mathsf{ages}}\, \langle \mathsf{Map}[\mathit{HitMiss} \mapsto \mathit{Value}] \rangle_{\mathsf{profile}}$$

$$\langle \mathit{Val} \rangle_{\mathsf{instrlen}}\, \langle \mathit{Addr} \rangle_{\mathsf{fstaddr}}$$

The ic cell keeps the instruction cache content as information of the form $\mathit{Addr} \mapsto$ $\texttt{iwrap}(\mathit{PC},\mathit{Instr})$, where *Addr* is the cache address that holds the instruction *Instr* at the program point *PC* in the program. A number of parameters such as cache size, cache line size, and associativity characterize the cache memories and are specified in the cell param. The cache size is the total number of bytes that can be stored. The cache line size is the number of bytes that can be transferred to and from the memory, in one step. The associativity describes the relation between cache lines and memory blocks. A memory block can reside anywhere in the cache, in a group of cache lines or in exactly one line. This leads to the standard terminology of fully associative caches, for the first case, *N*-associative caches for the second, with *N* the number of cache lines and the direct-mapped caches for the last case. We assume the particular case of a cache line and a memory block having the same size.

The instruction cache configuration, *ICConfig*, also has a special cell labeled repl that keeps the cache address where the actual replacement should take place. The implementation of two of the most popular replacement policies: FIFO (round-robin) and LRU are activated from using the external function `updAges`. One particular difference between these two is that the LRU policy changes the "age" attribute on both hits and misses, whereas the FIFO policy on misses only. For more insights on the cache replacement policies in the context of WCET analysis, we refer to [99].

The profile cell contains two counters to keep profiling information about the number of hits and misses, whereas the age cell maintains a single counter for the current age of the cache. The instrlen cell keeps the value of the instruction size, a parameter that is necessary when we use disassembled executable files. For a similar reason, the fstaddr cell holds the address of the first instruction in the program, which is also used to compute how the memory blocks are assigned to cache lines.

This instruction cache configuration consists of two sub-configurations, one inherited from the *Interface Module* and another which is specific to it. The cells params, instrlen, fstaddr and profile are common to both instruction and data cache specifications and belong to the structural module *Interface Module*. On the other hand, the actual instruction cache content, in ic cell, is part of the functional module *IC Module*. We conclude our discussion on the instruction cache configuration with an emphasis on the repl and ages cells. Using external functions to calculate the cache address where the replacement should take place. Similarly, in the ages cell we trigger the age update algorithm, based on a selected replacement policy.

We present, in Fig. 5.5, the rules for instruction cache hit, the first rule and cache miss, the second rule.

The rule [IC-HIT] captures the instruction cache hit case. The k cell processes the instruction request, `geti`(PC), and sends back to the language semantics the program counter incrementation request, `incPC` followed by the actual instruction *Instr*. The ic cell contains this particular instruction at address *CA*. The information at a cache address is represented using the wrapper `iwrap`, which consists of the instruction and

```
rule  [IC-HIT] :
   <k> geti(PC:#Int32) => incPC(PC) ~> Ins ...</k>
   <ic>... CAddr:#Int32 |-> iwrap(PC,Ins:Instr) ...</ic>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32
              pt |-> P:#Int32 ...</params>
   <profile>... hiti |-> (H:#Int32 => H +Int32 1) ...</profile>
   <ages>A:Map => updAges (A,wia(CAddr),NL,M,P) </ages>

rule  [IC-MISS] :
   <k> geti(PC:#Int32) => imiss(PC) ...</k>
    <ic> ICache:Map </ic>
    <params>... csize |-> NL:#Int32 ca |-> M:#Int32 ...</params>
    <fstaddr> FAddr:#Int32 </fstaddr>
    <instrlen> ILen:#Int32 </instrlen>
when notBool (inCache (ICache, PC, FAddr, ILen, NL, M))
```

Figure 5.4: 𝕂-rules for instruction cache behavior

the instruction address in the memory, as the value *PC*. Since our cache modeling is parameterized by the cache capacity `csize`, associativity `ca` and the type of the replacement policy `pt`, which are all kept in the cell param, the hit/miss decision is based on these parameters. The cell representing the age information is updated using the function `updAges`, which we explained in the previous sub-section. Also, the profiling information increments the number of instruction cache hits, the `hiti` value.

The first rule of the cache miss scenario, [IC-MISS], detects a miss when the instruction at program counter *PC* is not `found` in the instruction cache *ICache*, and the message `imiss`(*PC*) is sent to the *Main Memory Module*. This rule could rely on an offline computation of `checkCache`. The two special 𝕂 cells, fstaddr and instrlen provide additional parametrization when checking the content of the instruction cache. This additional information is with respect to the program layout in the main memory, and directly to the way the program instructions are mapped to the cache lines.

The third rule in Fig. 5.5, [IC-RETPOS], has the instruction fetched from main memory in the k cell, wrapped in a special message called `iret`, together with the instruction

```
rule  [IC-RETPOS] :
   <k> iret(PC:#Int32, Ins:Instr) => cplace(PC,Ins) ...</k>
   <ic> ICache:Map </ic>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32 ...</params>
   <repl> _ => retPos(A,PC,ICache,FAddr,ILen,M,NL) </repl>
   <ages> A:Map </ages>
   <fstaddr> FAddr:#Int32 </fstaddr>
   <instrlen> ILen:#Int32 </instrlen>

rule  [IC-REPLACE] :
   <k> (cplace(PC,Ins) => (incPC(PC) ~> Ins)) ...</k>
   <ic>... Pos |-> ( _ => iwrap(PC,Ins)) ...</ic>
    <params>... csize |-> NL:#Int32 ca |-> M:#Int32
           pt |-> P:#Int32 ...</params>
    <profile>... missi |-> (M1:#Int32 => M1 +Int32 1)
                    ...</profile>
    <repl> Pos:#Int32 </repl>
    <ages>A:Map => updAges (A,wia(Pos),NL,M,P) </ages>
```

Figure 5.5: 𝕂-rules for the replacement for the cache miss

address *PC*. This particular rule kicks in the two-step process of the instruction cache update. The repl cell uses an external function `retPos` to detect the correct location in the cache, where instruction *Ins* should be placed. `retPos` is also parametrized on both cache and program related parameters (which were introduced earlier). The term `cplace` is used as a separator between the two steps of the cache update algorithm.

The last rule, [IC-REPLACE], does the actual instruction cache update, as shown in the ic cell. In this rule *Pos* stands for the previously computed cache location, in the special cell repl. The instruction cache content ic stores the instruction *Ins* in this particular cache location, *Pos*. Again, the cache related parameters in the params cell are used in the age computation. Also, this rewrite rule counts a cache miss, in the cell profile.

We present a concise design of the instruction cache behavior, with a high degree of reusability, achieved using two special functions, `updAges` and `retPos`. These allow external computation to populate the special 𝕂 cells ages and respectively repl.

This modeling style allows major changes in the instruction cache behavior (i.e. new functionality for `updAges` and `retPos`), without affecting the four rewrite rules at all. One such change is the replacement policy.

5.3 Data Caches

The $\mathbb{K}$ definition of the data cache behavior follows the same modeling principles that we state in the last paragraph of the previous sub-section. The *DC Module* filters the data requests from the language semantics, via the load/store instructions. The initial design of the system, case (a) in Fig. 5.3, is extended to include the data cache memory as in the case (c). In this way, the input and output communication interfaces of the *Language' Semantics Module* and *Main Memory Module* handle new messages to read and write word integer values (i.e. `getd`, `putd`), double word integer values (i.e. `getDbl`, `putDbl`) and similar ones for single and double precision floating point values. The communication between the data cache module and the other modules is also realized via the `k` cell.

As in the case of the instruction cache models, there are several data cache specifications of interest. First, there is a stub implementation of a data cache memory that is useful to execute programs, where data is directly brought from the main memory. The entire set of communication messages with respect to data types forwards the requests from the *Language Semantics Module* to the *Main Memory Module*. The justification for the data cache stub module is also inspired from the unit testing methodology,

Second, there is a similar concise design of the data cache memory, as for the instruction cache. We follow the same principle of modularity using external functions such as the one for the cache replacement policy. The overall behavior of the data cache memory is complex, mostly because of the cache miss scenarios. In the case of the instruction cache behavior, the only cache miss case is when an instruction could not be found after a read request. In the case of the data cache behavior, along with the classical cache miss on data read, there is the cache miss on data write.

We explain next a cache miss on write scenario. Let us consider a 4-line fully associative cache with the following cache content: the memory blocks *a* to *d* reside in the cache lines 1 to 4, respectively, with the block *a* being the youngest and *d* the oldest. The main memory contains its own copies of the blocks *a* to *d*. If the program issues a request to modify data *e*, which is not found in the data cache memory, then it appears a problem of maintaining the consistency between the cache and main memory. There are the following two possibilities, named writing policies:

- write-through : data *e* is modified directly in the main memory (in the case of writing a new value for data *a*, which is found in the data cache memory, both the cache and the main memory are modified)
- write-back : data *e* is brought from the main memory into the data cache memory, then it is modified. In this case, the two copies of the same data *e* have different values, and the cache one is flagged, using a bit called dirty bit. Such a bit allows multiple modification of the data in the data cache, and when this data should be evicted, then the main memory copy of it is updated with the corresponding value from the cache.

We present an alternative design for data caches, one that relies on another abstraction level, corresponding to the information writing in the cache. The configuration of the data cache is:

$$DCConfig \equiv \langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[Addr \mapsto Instr]\rangle_{\mathsf{dc}}\ \langle \mathsf{Map}[DCParam \mapsto Val]\rangle_{\mathsf{param}}\ \langle Cnt\rangle_{\mathsf{ages}}$$

$$\langle Val\rangle_{\mathsf{repl}}\ \langle \mathsf{Map}[HitMiss \mapsto Value]\rangle_{\mathsf{profile}}\ \langle \mathsf{Map}[Addr \mapsto Bit]\rangle_{\mathsf{dbits}}$$

$$\langle Val\rangle_{\mathsf{instrlen}}\ \langle Addr\rangle_{\mathsf{fstaddr}}$$

We briefly explain the data cache configuration, which has a sub-configuration that is shared with the instruction cache configuration and is defined in the *Interface Module*. This particular sub-configuration consists of the following 𝕂 cells: `params` for the cache parameters, `profile` with the counters of the cache hits and misses, `instrlen` and `fstaddr` for the instruction length and respectively the memory address of the first instruction in the program.

The rest of the 𝕂 cells are specific to data caches. The `dc` cell has the content of the data cache memory, as a mapping from cache addresses to stored data (i.e. a wrapping `dwrap` with both data address in the main memory and the actual data). The two cells `ages` and `repl` have the same role as in the case of the instruction cache model: `ages` keeps information about the relative age of the cached data and `repl` keeps the cache address where the data replacement should take place. The `dbits` cell keeps a similar mapping as the `ages` cell, each cache line with inconsistent data with respect to the main memory is flagged using a so-called dirty-bit. We model the write-through and the write-back policies and the `dbits` cell is a stub (or not used) for the former policy and has the proper functionality for the latter policy.

We present next the 𝕂 rewrite rules for the data cache modeling. For presentation purposes, we split the specification into several sets of rules as follows: the hit rule and the miss on read case rules are represented in Fig. 5.7, the two hit on write cases (for write-back and write-through) cases in Fig. 5.8.

The first rule in Fig. 5.7, [DC-HITREAD], has on top of the `k` cell the data request from the formal semantics, via the communication message `getd`. This is the cache hit on read case, as the `dc` cell has at the cache address `CAddr` the requested data, in the `dwrap` term. The number of data cache hits, represented by `hitd`, is incremented in the `profile` cell. The age information is updated in the same way as for the instruction caches.

In the second rule [DC-MISSREAD], the same `getd` data request from the cache

```
rule [DC-HITREAD] :
   <k> getd(Addr:#Int32) => Data ...</k>
   <dc>... CAddr:#Int32 |-> dwrap(Addr,Data:#Int32) ...</dc>
   <profile>... hitd |-> (H:#Int32 => H +Int32 1) ...</profile>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32
                         pt |-> P:#Int32 ...</params>
   <ages>A:Map => updAges (A,CAddr,NL,M,P) </ages>

rule [DC-MISSREAD] :
   <k> getd(Addr:#Int32) => dmiss(Addr) ...</k>
   <dc> DCache:Map </dc>
   <params>...  csize |-> NL:#Int32 ca |-> M:#Int32
                          pt |-> P:#Int32 ...</params>
when notBool (inCache (DCache, Addr, NL, M))
```

Figure 5.6: 𝕂-rules for data cache behavior - the cache hit and miss on data read

is transformed into `dmiss`, the same data request sent to the *Main Memory Module*. This rewrite rule is conditional, the function `inCache` checks if the data cache `DCache` contains the requested data.

The two rewrite rules in Fig. 5.7, [DC-DRET] and [DC-DPLACE] implement the two phase replacement algorithm, in a similar fashion with the rules [IC-RETPOS] and [IC-REPLACE] in Fig. 5.5. The only difference is in the profile cell, where the number of data cache misses `missd` is incremented. This set of four rules forms a definitional template in our data cache modeling. Since there are several types of data request (i.e. double words, floats), there are instances of this template for these types. A similar reasoning is applied for write requests, with `putd`-like communication messages on top of the k cell.

The rewrite rules in Fig. 5.8 show the hit on write cases. The two rules [DC-HITWRWB] and [DC-HITWRWT] focus on the write-back and respectively on the write-through writing policies. We start with the former, the rule [DC-HITWRWB]. The communication message `putd` requests `Data` from the specified memory address `Addr`. In the data cache, this particular information is at the cache address `Pos`. One particularity

```
rule [DC-DRET] :
   <k> dret(Addr:#Int32, Data:#Int32) =>
           dplace(Addr,Data) ...</k>
   <dc> DCache:Map </dc>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32
                         pt |-> P:#Int32 ...</params>
   <repl> _ => retPos(DCache, Addr, NL, M) </repl>

rule [DC-DPLACE] :
   <k> dplace(Addr,Data) => Data ...</k>
   <dc>... Pos |-> ( _ => dwrap(Addr,Data)) ...</dc>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32
                         pt |-> P:#Int32 ...</params>
   <profile>... missd |-> (M1:#Int32 => M1 +Int32 1)
                       ...</profile>
   <repl> Pos:#Int32 </repl>
   <ages>A:Map => updAges (A,Pos,NL,M,P) </ages>
```

Figure 5.7: 𝕂-rules for data cache behavior - the replacement for the cache hit on read case

of this writing policy is that, in the case of a cache hit, the data is modified in the `dc` cell, while, in the `k` cell, the writing request is dissolved. The cache hit on write is transparent for the *Main Memory Module*. The particular data cache address `Pos` is flagged, in the `dbits` cell.

In the case of a write-through policy, the rule [DC-HITWRWT], there are several important changes. The data write request `putd`, in the `k` cell, is forwarded to the *Main Memory Module*, using a communication message called `write`. The third parameter of `write` represents an identifier for the writing policy, with the value 0 for the write-back and the value 1 for the write-through policies. The writing policy type is also a parameter of our data cache modeling, therefore it is placed in the `params` cell, as the value of `wp`, along with the structural characteristics of caches. With respect to the data types that are manipulated in the *SSRISC* programs, the two rules in Fig. 5.8 are word-size instances of a more general template. There are also similar 𝕂 rewrite rules for the double word or various floating point representations of a data.

```
--- write (addr, data, 0) is for write back
rule [DC-HITWRWB] :
   <k> putd(Addr:#Int32,Data:#Int32) => . ...</k>
   <dc>... Pos:#Int32 |-> (dwrap(Addr,_) =>
                dwrap(Addr,Data)) ...</dc>
   <profile>... hitd |-> (H:#Int32 => H +Int32 1) ...</profile>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32
                 pt |-> P:#Int32 ...</params>
    <ages>A:Map => updAges (A,Pos,NL,M,P) </ages>
    <dbits>... Pos |-> (_ => 1) ...</dbits>


rule [DC-HITWRWT]
   <k> putd(Addr:#Int32,Data:#Int32) =>
           write(Addr,Data,W) ...</k>
   <dc>... Pos:#Int32 |-> dwrap(Addr,Data) ...</dc>
   <profile>... hitd |-> (H:#Int32 => H +Int32 1) ...</profile>
   <params>... csize |-> NL:#Int32 ca |-> M:#Int32
                 pt |-> P:#Int32 wp |-> W:#Int32 ...</params>
   <ages>A:Map => updAges (A,Pos,NL,M,P) </ages>
```

Figure 5.8: 𝕂-rules for data cache behavior - hit on write

We discuss next the cache miss on write scenarios. In such cases, a write-through policy allows two possible scenarios: first, the data is directly written to the main memory, without getting it into the cache memory, while second, the data is loaded into the cache and then modified. The writing policy is, in this first case, with no-allocation, while in the second case it is with-allocation. We implement the traditional write-through with no-allocation and the write-back with-allocation.

5.4 Related Work

Research on cache modeling for the WCET analysis purposes focuses on elements of the cache behavior that affects the most the execution time of a program. Naturally, the main target is the classification of cache hits and misses, projected on the time spent and penalty incurred, in relation with the program instruction and data.

One of the first approaches to consider cache memory models for WCET analysis is in [76]. It is a integer linear programming (ILP) model, with the following general idea in mind: to group program's instructions with respect to the cache parameters such that all the instructions in a group behave in the same way with respect to the cache interaction. The ILP objective function is then modified to integrate this refinement, for example the execution count of an instruction is equal to the number of executions when the instruction is a cache hit plus the number of executions when the instruction is a cache miss. These cache constraints address, via integer variables, two situations - of conflicting and non-conflicting cache blocks. To be more specific, when a particular instruction is fetched in the code, the others from the same block will be fetched as well. Therefore, the sum of these instructions' cache misses is at most one, because only one instruction is specifically classified as miss. This ILP based modeling of caches has scalability issues.

A similar cache hit-miss classification is achieved with the abstract interpretation (AI) approaches [108, 111]. A fixpoint procedure on the control flow graph of the program computes cache related invariants to each program point. AI-based models for caches are integrated in `aiT` timing analyzer, a tool which achieved industrial success. We elaborate more on the AI-based approaches in the subsequent Section 6.

We model the instruction and data caches with the following trade-off in mind: to execute programs and to facilitate development of analysis tools over the models. Both ILP and AI models we mentioned in the previous paragraphs are specifically designed for analysis purposes. In comparison, our $\mathbb{K}$ configuration for the caches includes two cells, `ic` and `dc` for the actual instruction and respectively data cache content, using the the two wrappers `iwrap` and `dwrap`. None of the two approaches explicitly consider the actual instruction in the modeling, its program counter suffices for the classification into cache hits and misses.

The interest in cache models for timing analysis is beyond the split case of instruction and data caches. For example, shared caches, which are present in multicore embedded systems are addressed in [117]. Here, the inter-core conflicts for cache accesses adds a new layer of complexity to the modeling, as it is the potential multi-level structure of the cache memories. In this paper, there are two cache levels, a private L1 split cache (instruction and data) and a shared L2 cache. The approach looks for a similar classification of instructions with respect to their cache activity. Because of the multi-levels, there are several more cases such as: hit in the L1 cache, miss in the L1 and hit in L2 cache, miss in both levels. All these increase the complexity at the modeling level. There is an additional complexity, because of the concurrent nature of the program (the threads implant a time stamp on the cache activity, now it is important to know when a particular cache request was issued).

Our solution for instruction and data cache modeling is modular and, using the communication interfaces (input and output) of each module, it allows extensions. Therefore, with respect to the previous work, in [117], our model corresponds to the private L1 caches, and a new L2 shared cache module require changes at the level of *Interface Module*, with the new communication messages between L1 and L2 levels. Also, there are changes at the level of specification, with a high degree of reusability for the L1 caches and a complete new implementation for the shared L2 cache.

There are also other hardware related issues, addressed in the context of WCET analysis. While we omit the pipeline models, we mention prefetching in [116] and cache hierarchies are studied in [49] or in [50].

Chapter 6

Control-flow Abstractions for WCET Analysis

Our modular design consists of the formal executable semantics of *SSRISC* and the specification of caches and main memory systems. We design these two components to help implement abstractions in an incremental way. The Fig.3.9, in Section 3 shows the two directions in analysis and verification tools development. These can be used on our $\mathbb{K}$ definition to explore the state space in an explicit (i.e. model checking) or an implicit (i.e. abstract interpretation) way.

In this section we present how to define abstractions, starting with our proposed modular design of an embedded system (both hardware and software components). We start with a popular abstract representation of a program is the control flow graph (CFG), which, in the case of executables, is a tricky problem (due to the indirect jumps). While we do not approach all the corner cases that make the CFG construction difficult, we illustrate the methodology first using reachability analysis to collect control-flow edges (in the next Section, 6.1). A major issue in the program analysis and verification, in general is to discover the loops' behavior. If we project this on the WCET analysis methods, the timing behavior of the loops should be either automatically determined, or manually provided. The former is desirable, but difficult to achieve, while the second is accepted by the community. The Section 6.2 presents a $\mathbb{K}$ based annotation style together with an unfolding procedure, which generates all the loop executions under the specified constraints [5]. This approach could be used to generate more accurate information to tighten the timing prediction of loops.

6.1 Abstractions for CFG Extraction

Usually, the low-level WCET estimation is transformed into the path analysis and processor behavior analysis. The former works with the control flow graph of the program and, via clever abstractions, attempts to eliminate infeasible paths. One of the strengths of our approach is the possibility of using the concrete formal executable semantics to derive such abstract semantics. There are two possible ways to manipulate the concrete definition, for both hardware and software components: through local changes of the semantics rules or leaving the specification unmodified.

Our approach works on an assembly programs, obtained from disassembling the Simplescalar [18] executable files. One of the problems with the executables is that each instruction address is a potential target of an indirect jump, represented in our language by the `jr` instruction. The actual target address is known precisely at runtime. We define the program unfolder in two phases. In the first phase, the modular definition of the formal executable semantics is extended to accommodate control flow information. The only modified semantic rules are those of branch and jump instructions. The second phase uses the over-approximation of the control-flow graph, computed during the first phase and a set of user loop bound annotations. The concrete semantics of the language is used to define the simplified abstract semantics that uses symbolic values instead of real values. The definitional program unfolder outputs a trace semantics for the program. Our method targets a particular class of hard real-time programs, which have a bounded number of loops iterations and recursive function calls.

We also rely on several assumptions. The analyzed code is structured, to ensure a well-specified unfolding of the program. This assumption triggers another one, in the case of an indirect jump, the instruction address that causes the jump is eliminated from the set of possible targets, to not introduce infinite executions. Also, the target for the indirect jump set of potential addresses is limited to only the addresses that are in the same block of instructions as the indirect jump instruction. The current design and implementation is amenable to further extensions.

We present next an abstraction for CFG extraction, using local modifications of the *SSRISC* language semantics. This abstraction relies on a simple data abstraction, which is presented in Chapter 7. We give a short overview on this data abstraction, then we present the configuration and the $\mathbb{K}$ rewrite rules for the CFG construction.

We consider the case where all the input program variables are initialized with an unknown value. The abstract semantics is derived out of the concrete semantics. Since the program manipulates an abstract value, the domain of possible values for the program variables is extended with this special value. This action *keeps all the* $\mathbb{K}$ *semantics rules unchanged*, but triggers further extensions in the support operations. For example, the addition between two concrete values is extended to handle the symbolic abstract value such that, if any of the operands is symbolic, the result is symbolic. This abstraction enables all the possible executions, as the $\mathbb{K}$ rewrite rules for the branch instructions rely on a comparison between, potentially symbolic values.

The abstract execution of the *SSRISC* semantics, in the *Language Semantics Module*, for the CFG extraction uses the previous one and requires extensions of the concrete semantics rules, as well. Since, the $\mathbb{K}$ framework is specialized on design of programming languages, the extension of this analysis is projected first at the level of configuration. Hence, there is a new cell, called `cfg` that holds the (abstract) control flow graph. The elements of `cfg` are edges of the form $src \mapsto dst$, where src and dst represents the source

$$\text{RULE: } \left\langle \frac{\texttt{beq}\ V_1, V_2, Addr;}{\texttt{setPC(Bool2Int(}V_1 ==_{BoolSy} V_2\texttt{)}, Addr\texttt{)}} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \left\langle \cdots \frac{\cdot}{PC -_{Int32} 4 \mapsto Addr} \cdots \right\rangle_{\mathsf{cfg}}$$

$$\text{RULE: } \left\langle \frac{\texttt{bne}\ V_1, V_2, Addr;}{\texttt{setPC(Bool2Int(}V_1 =/=_{BoolSy} V_2\texttt{)}, Addr\texttt{)}} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \left\langle \cdots \frac{\cdot}{PC -_{Int32} 4 \mapsto Addr} \cdots \right\rangle_{\mathsf{cfg}}$$

$$\text{RULE: } \left\langle \frac{\texttt{j}\ Addr;}{\texttt{setPC(}1, Addr\texttt{)}} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \left\langle \cdots \frac{PC -_{Int32} 4 \mapsto PC}{PC -_{Int32} 4 \mapsto Addr} \cdots \right\rangle_{\mathsf{cfg}}$$

$$\text{RULE: } \left\langle \frac{\texttt{jr}\ Rs;}{\texttt{setPC(}1, Rs\texttt{)}} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \left\langle \cdots \frac{PC -_{Int32} 4 \mapsto PC}{PC -_{Int32} 4 \mapsto \mathsf{symb}} \cdots \right\rangle_{\mathsf{cfg}}$$

Figure 6.1: CFG Extraction Rules

and destination program points. The new abstract configuration, $Config_{CFG}$ is presented next.

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

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

$$\langle \mathsf{Map}[Int32 \mapsto Int32] \rangle_{\mathsf{cfg}}$$

Based on the execution of the first simple abstract semantics, the `cfg` cell is updated with edges, according to the rules in Fig.6.1. We take advantage of the $\mathbb{K}$ framework modularity and design this set of rules to minimize the number of changes in the semantic rules. In this way, out of all the *SSRISC* instructions, only the jump and branch instructions alter the normal program flow and require different manipulation. We handle the default case of going from a program point to the next with only one $\mathbb{K}$ rule, not shown in Fig.6.1, when we add the corresponding edge in the `cfg` cell, during instruction fetch.

Since this is not the only possible outcome, for branch instructions, nor the correct one, for jump instructions, the four rules for the over-approximated CFG add the necessary edges. For example, when the cell `k` contains the `beq` instruction, the cell `cfg` gets the "branch-taken" edge, $PC \mapsto Addr$. Also, in case of a `j` instruction, the edge from the current *PC* to the target address replaces the default edge. In case of an indirect jump, instruction `jr`, the actual value of the target address is known at runtime, therefore, we use the symbolic value `symb`. The set of all possible target addresses is the set of all program points, excluding, by assumption, the address of that particular `jr` instruction. We obtain the set of all possible edges, meaning the over-approximated CFG, via reachability analysis over the program, performing the union of `cfg` cell contents of all possible execution traces.

At the level of module representation, the abstraction which is partially presented in Fig.6.1, requires, besides the modification of the *Builtin Module*, a new implementation of the *Language Semantics Module*. The extent of this modification is only for the $\mathbb{K}$ rewrite rules for branch and jump instructions, with the updates in the `cfg` cell.

6.2 Definitional Unfolder

The results of a WCET analysis should be tight, when are reported to a measured time of the worst case scenario. A major obstacle is to analyze the program loops' behavior, from a timing perspective. Thus, program assertions are indispensable in certain situations, and, in our target application, these assertions are represented by loop bounds.

The unfolding uses the computed CFG (described in the previous section), together with manual annotations for loop bounds. We represent the annotations in a $\mathbb{K}$ cell, called loops and a loop stack cell, lstack, to keep track of the current state of the unfolding. The unfolder configuration is:

$$UfldConfig \equiv Config_{CFG} \ \langle \mathsf{Map}[PC \mapsto Loop] \rangle_{\mathsf{loops}}$$

$$\langle \mathsf{List}(Int32, Int32) \rangle_{\mathsf{lstack}}$$

The loops cell contains all the loops in the program, together with their respective bounds. We represent each loop as a tuple, with bound information and the following program points of interest: the first and the last executed instructions in the loop and the first instruction that is executed when we exit the loop. The unfolding procedure relies on this particular definition of loop information. The `lstack` cell uses a list to simulate a stack behavior that has on top the current unfolded loop, together with the remaining number of iterations. The rules for the CFG-based unfolder are in Fig.6.2 and in Fig.6.3. We use three predicates: `notIn (CFG,PC)` checks if the *CFG* contains a loop starting at *PC*, `noBnd(PC,Stack)` checks if *PC* is in the top of the *Stack*, while `chkFst(PC,CFG)` returns true if the *PC* is the first instruction in the program.

Let us consider a program that has a loop starting at program point 1, the jump back instruction (i.e. the last instruction in the loop body) is at program point 5 and the first instruction that is executed when the flow exists the loop is at program point 6. And let us assume that this loop is executed less than 10 times. Then, we represent the loop, in the cell `loops`, as $1 \mapsto loop(5,6,10)$. This kind of notation permits jump backs at different addresses, and, in this way to uniquely identify loops with the same starting point.

All the program unfolder rewrite rules fetch a new instruction, using the term `geti(PC)`, depending on loop nesting and bound values. At the level of module representation, this communication involves the *Language Semantics Module* has the communication message `geti` in its input interface and the *Main Memory Module*, which is capable of receiving it. The implementation of the program unfolder is in a new module, called *Unfold Module*, which communicates with *Main Memory Module*, and substitutes the entire language semantics.

The first three rewrite rules, in Fig.6.2, cover the simple case of programs without loops, rule [CFG-INIT], the first instruction in programs with loops, rule [CFG-NOFST], respectively an instruction that is not the first nor the last executed in the loops, rule [CFG-NOLST]. When the current executed instruction is also a loop test, we distinguish

RULE [CFG-INIT]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle \cdot \rangle_{\mathsf{loops}}$$

RULE [CFG-NOFST]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle PC \mapsto PC_2\ CFG \rangle_{\mathsf{cfg}} \langle L \rangle_{\mathsf{loops}}$$

when $\texttt{chkFst}(PC, CFG) \wedge_{Bool} \texttt{noBnd}(PC, L)$

RULE [CFG-NOLST]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle PC_2 \mapsto PC\ CFG \rangle_{\mathsf{cfg}} \langle L \rangle_{\mathsf{loops}}$$

when $\texttt{notIn}(CFG, PC_2) \wedge_{Bool} \texttt{noBnd}(PC, L)$

RULE [CFG-FSTLOOP]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle \cdots PC \mapsto \texttt{loop}(\_,\_,B) \cdots \rangle_{\mathsf{loops}} \left\langle \frac{\cdot}{(PC, B)} \right\rangle_{\mathsf{lstack}}$$

Figure 6.2: CFG-based Unfolding Rules - init stages

several situations.

First, in rule [CFG-FSTLOOP], in Fig.6.2, the unfolding process reaches the first loop, pushing the current loop start instruction and its bound onto the stack. Rule [CFG-ITER1] covers the case of getting to an inner loop and pushing into the stack its starting program point and the number of iterations, while having more iterations for the current loop unfolding is in rule [CFG-ITER2]. The unfolding process could exit a particular loop at any iteration, and rule [CFG-EXITL] emphasizes this case by popping the loop stack. The last two rules present the jump back caused by last executed instruction of a loop, rule [CFG-LOOPJB], respectively the execution of the instruction in the loop that follows immediately after the loop test, rule [CFG-LOOPFT]. The number of remaining iterations for this particular loop is decremented during this step.

To obtain a safe WCET bound for a given program, we need to consider all the possible program executions, implicitly or explicitly. The transition system that a rewrite theory provides could be unfolded using the special `search` command. The class of hard real-time programs that we consider as input, assume that a program terminates and, RULE [CFG-ITER1]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle \cdots PC \mapsto \texttt{loop}(\_,\_,B) \cdots \rangle_{\mathsf{loops}} \left\langle \frac{(PC_1,B_1)}{(PC,B)\ (PC_1,B_1)} \cdots \right\rangle_{\mathsf{lstack}}$$

when $PC_1 \neq_{\mathit{Int32}} PC$

RULE [CFG-ITER2]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle \cdots PC \mapsto \texttt{loop}(\_,\_,\_) \cdots \rangle_{\mathsf{loops}} \langle (PC,B) \cdots \rangle_{\mathsf{lstack}}$$

when $B \geq_{\mathit{Int32}} 0$

RULE [CFG-EXITL]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC_1)} \right\rangle_{\mathsf{k}} \langle PC_1 \rangle_{\mathsf{pc}} \langle \cdots PC \mapsto PC_1\ PC \mapsto PC_2 \cdots \rangle_{\mathsf{cfg}}$$

$$\langle \cdots PC \mapsto \texttt{loop}(\_,PC_2,\_) \cdots \rangle_{\mathsf{loops}} \left\langle \frac{(PC,B)}{\cdot} \cdots \right\rangle_{\mathsf{lstack}}$$

when $B >_{\mathit{Int32}} 0$

RULE [CFG-LOOPJB]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC_2)} \right\rangle_{\mathsf{k}} \langle PC_2 \rangle_{\mathsf{pc}} \langle \cdots PC \mapsto \_\ PC \mapsto PC_2 \cdots \rangle_{\mathsf{cfg}}$$

$$\langle L\ PC \mapsto \texttt{loop}(\_,PC_2,\_) \rangle_{\mathsf{loops}} \langle (PC,B) \cdots \rangle_{\mathsf{lstack}}$$

when $B >_{\mathit{Int32}} 0 \wedge_{\mathit{Bool}} PC_2\ \texttt{notIn}\ L$

RULE [CFG-LOOPFT]:

$$\left\langle \frac{\cdot}{\texttt{geti}(PC)} \right\rangle_{\mathsf{k}} \langle PC \rangle_{\mathsf{pc}} \langle \cdots PC_1 \mapsto \texttt{loop}(PC,\_,\_) \cdots \rangle_{\mathsf{loops}} \left\langle \frac{(PC_1,B)}{(PC_1,B -_{\mathit{Int32}} 1)} \cdots \right\rangle_{\mathsf{lstack}}$$

when $B >_{\mathit{Int32}} 0$

Figure 6.3: CFG-based Unfolding Rules - entry in and exit from a loop

in our formal semantics, presented in Chapter 4, we denote the final computational task with the token `last`. Therefore, all the program executions should terminate in a state that has `last` in the *k* cell. A single program execution can terminate in several ways, we present next the two such situations that are covered by our *SSRISC* language definition. First, there is the normal termination, when there are no more instructions to be executed. Second, there is the error termination due to overflow of arithmetic addition/subtraction or due to division by zero. Both situations are handled via pseudo-instructions that generate the `break` instruction.

We employ the reachability analysis, using the `search` command provided by

Maude system, for the state that has `last` as the current computational task. The timing information is updated along each execution path and, when we terminate the path exploration, the WCET is the maximum of these computed time values. Since we work on straight-line hard real-time programs, the state space exploration guarantees to terminate. We rely on the reachability analysis to determine both the control flow graph generation and the set of all possible program executions.

6.3 Related Work

In the context of control-flow graph extraction methodologies, [65] proposes an abstract interpretation based framework that relies on the notion of a partial control flow graph. The over-approximation of the CFG is based on combination of two collecting semantics: one for graph edges (control) and one for register values (data). [60] decomposes the program into basic blocks and procedures, ruling out, by assumption, the case of ovelapping entry and exit points. Also, the indirect jumps are explicitly labeled for all possible targets. Our approach also adheres to these assumptions. [8] considers an abstract domain of value sets and performs a data analysis to compute relations between possible values. All these works rely on ad-hoc encodings of the language semantics and propose abstract-interpretation based solutions to detect edges in the CFG. Therefore, in comparison to our proposal, their main strength is the data analysis to refine the set of target addresses of indirect jumps, whereas in our case it is the language definition, which is executable and reliable, assuming extended testing.

By program unfolding we understand a technique that generates all execution paths, and is an extension of the classical loop unfolding optimization in compilers. The set of program paths could contain particular infomation and be useful in various software development stages. We mention applications of program unfolding in profiling, as described in [118] and in verification, [38], as behavior of the initial system is behaviorally equivalent to its unfolded variant. Our unfolder generates execution paths having the entire state of the program, together with the timing information.

Chapter 7

Data-flow Abstractions for WCET Analysis

A WCET analysis is usually split into a program-related path analysis and a processor behavior analysis. The former requires accurate flow information, which can fall into two important categories, with respect to their applicability in timing analysis: automatic derivation of loop bounds and detection of infeasible paths. The aggregation-based abstractions, for example the ILP formulation, require knowledge of loop bounds to successfully collapse similar execution paths (i.e. loop iterations). Therefore, knowing loop bound is necessary in the process of timing analysis. On the other hand, computing timing bounds without knowledge about the infeasible paths in a program is possible, but the results may not be accurate. These two applications where data flow analyses help the WCET analysis apply on a target program. With respect to the underlying architecture models, there is another applications for data flow analyses. For example in data cache behavior abstractions, where data addresses should be computed in order for further analyses to apply.

To obtain a safe WCET bound for a given program, we need to consider all the possible program executions, implicitly or explicitly. We investigate $\mathbb{K}$ encodings of abstractions for WCET analysis for a special class of hard real-time programs, with the assumption that programs terminate. The formal executable semantics of *SSRISC*, in *Language Semantics Module*, contains all the necessary information to develop abstractions out of it. In this case, we concern with the semantics information with respect to program termination. The *SSRISC* semantics relies on a special token `last` to signal the final computational task and therefore, all the programs terminate in a state that has `last` in `k` cell. A program execution can terminate in several ways, we present next two such situations: the normal termination and the error termination (i.e. due to overflow of arithmetic addition/subtraction or due to division by zero). In Section 7.1 we present the problem of detection a particular case of infeasible paths, called erroneous paths, by executing the semantics [7].

In the context of WCET analysis, there are two data analyses singled out [115] as the most important ones: the constant propagation and the interval analysis. The reason is that these are at the base of more sophisticated analyses at both the level of program and architecture behavior prediction. In Section 7.1 we present an immediate implementation of a constant propagation, directly over the concrete semantics, without modifying it. The encoding is only at the level of the supporting operations, with a new *Builtin Module*, because the underlying lattice is flat. On the other hand, Section 7.2 presents an encoding of the standard interval analysis. This analysis requires modifications of the support operations, again a new *Builtin Module* and the new, abstraction specific module *Interval Abstr Module* containing the join operations.

7.1 Erroneous Paths Detection

The path analysis classifies the execution paths into feasible and infeasible. The programs may also exhibit *error execution paths*, either for certain input value or under special conditions (i.e. linking of recompiled code fragments). The most common errors have numerical causes such as overflow/underflow and division by zero, or are memory-related, in case of misaligned accesses. It is important to discover and to use error-related knowledge about programs to improve the timing predictability [109]. For example, the single-path programming technique [96, 66] advocates for predicated code generation, when division by zero errors are possible. Timing analyzers further utilize this predicated instrumentation to improve on timing bounds estimation. It is also possible that the underlying compiler generates preventive code to test if certain numerical errors are possible. The extra tests in the generated code implicitly cover the erroneous paths during the WCET analysis.

There are cases when undetected error paths lead to overestimation of timing bounds. We consider the simple, straight line assembly program, in Figure 7.2, where, for simplicity, we assume that each instruction executes in one time unit. All the registers, with the exception of `r0`, which has value 0, are initialized to symbolic values (i.e. a special value called `smb`) to exhibit all program behaviors, in this particular case, four possible executions. The longest executable path appears to be when both branches are not taken, which is actually an erroneous path, due to a division by zero at instruction 8. This comes after the following instructions: at program point 1, the register `r2` gets value 1, at line 3, register `r3` is updated with value `1`, which at line 7 is overwritten with value 0. The division raises an error, the execution stops before the end of the program and the path should be labeled as incorrect, and therefore removed as being the longest executable path.

An example of the error path detection via semantics executions is in Fig. 7.1.

In this particular section, we rely on $\mathbb{K}$ versatility to define both the programming language and associated analysis methods for erroneous paths detection in WCET estimation. Definitions of certain instructions explicitly state program error conditions such as integer overflow/underflow, division by zero or misalignment. In this way, the erroneous paths are explicitly exposed by the semantic rules. Using the formal

$$
\begin{array}{ll}
 & \langle \cdots \langle . \rangle_{\mathsf{k}} \, \langle 1 \rangle_{\mathsf{pc}} \, \langle \mathtt{r0}{\mapsto}0 \; \mathtt{r1}{\mapsto}\mathit{smb} \; \mathtt{r2}{\mapsto}\mathit{smb} \; \mathtt{r3}{\mapsto}\mathit{smb} \; \mathtt{r4}{\mapsto}\mathit{smb} \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{getPC(1)} \rangle_{\mathsf{k}} \, \langle 1 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{incPC(1)} \curvearrowright \mathtt{addi\ r2,r0,1;} \rangle_{\mathsf{k}} \, \langle 1 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{addi\ r2,r0,1;} \rangle_{\mathsf{k}} \, \langle 2 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r0}{\mapsto}0 \; \mathtt{r2}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\xrightarrow{\mathit{strict}} & \langle \cdots \langle \mathtt{addi\ r2,0,1;} \rangle_{\mathsf{k}} \, \langle 2 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r0}{\mapsto}0 \; \mathtt{r2}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{updateReg(}0 +_{\mathit{Int}32} 1\mathtt{,r2)} \rangle_{\mathsf{k}} \, \langle 2 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r2}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle . \rangle_{\mathsf{k}} \, \langle 2 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r2}{\mapsto}1 \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{getPC(2)} \rangle_{\mathsf{k}} \, \langle 2 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{incPC(2)} \curvearrowright \mathtt{beq\ r1,r0,5;} \rangle_{\mathsf{k}} \, \langle 2 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{beq\ r1,r0,5;} \rangle_{\mathsf{k}} \, \langle 3 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r0}{\mapsto}0 \; \mathtt{r1}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\xrightarrow{\mathit{strict}} & \langle \cdots \langle \mathtt{beq}\ \mathit{smb}\mathtt{,0,5;} \rangle_{\mathsf{k}} \, \langle 3 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{setPC(}\mathsf{Bool2Int}(\mathit{smb} ==_{\mathit{Bool}} 0)\mathtt{,5)} \rangle_{\mathsf{k}} \, \langle 3 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{setPC(}\mathsf{Bool2Int}(\mathit{true})\mathtt{,5)} \rangle_{\mathsf{k}} \, \langle 3 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{setPC(1,5)} \rangle_{\mathsf{k}} \, \langle 3 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle . \rangle_{\mathsf{k}} \, \langle 5 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{getPC(5)} \rangle_{\mathsf{k}} \, \langle 5 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{incPC(5)} \curvearrowright \mathtt{add\ r3,r2,r0;} \rangle_{\mathsf{k}} \, \langle 5 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{add\ r3,r2,r0;} \rangle_{\mathsf{k}} \, \langle 6 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r0}{\mapsto}0 \; \mathtt{r2}{\mapsto}1 \; \mathtt{r3}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\xrightarrow{\mathit{strict}} & \langle \cdots \langle \mathtt{add\ r3,1,0;} \rangle_{\mathsf{k}} \, \langle 6 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r0}{\mapsto}0 \; \mathtt{r2}{\mapsto}1 \; \mathtt{r3}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{ovf(1,0)} \curvearrowright \mathtt{updateReg(}1 +_{\mathit{Int}32} 0\mathtt{,r3)} \rangle_{\mathsf{k}} \, \langle 6 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r3}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{updateReg(}1 +_{\mathit{Int}32} 0\mathtt{,r3)} \rangle_{\mathsf{k}} \, \langle 6 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r3}{\mapsto}\mathit{smb} \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle . \rangle_{\mathsf{k}} \, \langle 6 \rangle_{\mathsf{pc}} \, \langle \mathtt{r0}{\mapsto}0 \; \mathtt{r1}{\mapsto}\mathit{smb} \; \mathtt{r2}{\mapsto}1 \; \mathtt{r3}{\mapsto}1 \; \mathtt{r4}{\mapsto}\mathit{smb} \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\cdots \to & \langle \cdots \langle . \rangle_{\mathsf{k}} \, \langle 8 \rangle_{\mathsf{pc}} \, \langle \mathtt{r0}{\mapsto}0 \; \mathtt{r1}{\mapsto}\mathit{smb} \; \mathtt{r2}{\mapsto}1 \; \mathtt{r3}{\mapsto}0 \; \mathtt{r4}{\mapsto}\mathit{smb} \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{getPC(8)} \rangle_{\mathsf{k}} \, \langle 8 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{incPC(8)} \curvearrowright \mathtt{div\ r2,r3;} \rangle_{\mathsf{k}} \, \langle 8 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{div\ r2,r3;} \rangle_{\mathsf{k}} \, \langle 9 \rangle_{\mathsf{pc}} \, \langle \cdots \mathtt{r2}{\mapsto}1 \; \mathtt{r3}{\mapsto}0 \cdots \rangle_{\mathsf{regs}} \cdots \rangle_{\mathsf{T}} \\
\xrightarrow{\mathit{strict}} & \langle \cdots \langle \mathtt{div\ 1,0;} \rangle_{\mathsf{k}} \, \langle 9 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{div0(0)} \curvearrowright \mathtt{updateLo(}1/_{\mathit{Int}32}0\mathtt{)} \curvearrowright \mathtt{updateHi(}1/_{\mathit{Int}32}0\mathtt{)} \rangle_{\mathsf{k}} \, \langle 9 \rangle_{\mathsf{pc}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{break;} \rangle_{\mathsf{k}} \, \langle 9 \rangle_{\mathsf{pc}} \, \langle 0 \rangle_{\mathsf{break}} \cdots \rangle_{\mathsf{T}} \\
\to & \langle \cdots \langle \mathtt{last} \rangle_{\mathsf{k}} \, \langle 9 \rangle_{\mathsf{pc}} \, \langle \cdots 1 \cdots \rangle_{\mathsf{break}} \cdots \rangle_{\mathsf{T}}
\end{array}
$$

Figure 7.1: Error path detection via semantics execution

semantics, we do not rely on the compiler to generate preventive code, nor on manual instrumentation for error path detection. We use the concrete executable semantics,

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

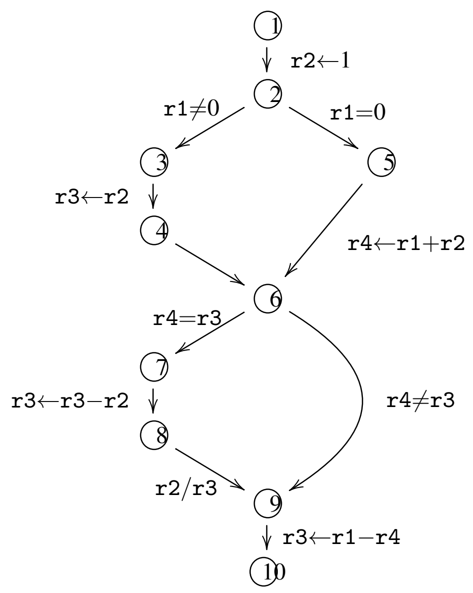


Figure 7.2: *SSRISC* program (left) with control flow graph (right)

augmented with timing information, to derive abstract semantics and then we employ reachability analysis techniques to detect and eliminate erroneous paths in the context of WCET analysis.

The $\mathbb{K}$-semantic rules of some *SSRISC* instructions, i.e. `add` and `div` embed tests to prevent numerical errors such as overflow and division by zero, respectively. The language definition also poses memory-related error checks such as misaligned data accesses, for a double word load instruction.

The constant propagation, along with the interval analysis are among the most used static analyses, in the context of WCET analysis [115]. In this section we discuss how the execution of the constant propagation is integrated in our framework, while in Section 7. 2 we present the interval analysis in the same context.

A constant propagation analysis produces, at each program point in the program, the set of variables (in our case registers) which have constant values. Therefore, the underlying lattice is flat, with the bottom element the value which is not a constant and the top the value which could be a constant. According to the general scheme for an abstract interpretation based analysis, presented in Section 2.4, 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. Our approach towards semantics executions using symbolic unknown value is meant for the erroneous path detection via reachability analysis.

Next, we describe how abstract semantics for error paths detection can be obtained from the language definition. The first step is to consider the input program variables initialized with an unknown value, that we name *symb*. This is the constant propagation analysis that we generically referred to in the previous paragraph. While the *Language Semantics Module* remains unmodified, the abstraction triggers extensions in the support operations. A newly defined module for symbolic 32-bit integer operations replaces the corresponding *Builtin Module* used by the concrete semantics. For example, the addition operation between two concrete values, denoted by $+_{Int32}$, is extended to handle *symb* abstract value such that, if any of the operands is symbolic, the result is symbolic. This abstraction indeed enables all the possible executions, as the $\mathbb{K}$ rewrite rules for the branch instructions handle potentially symbolic values.

The program in Figure 7.2 has four execution paths, all of them could be exercised under appropriate input data. There are two paths having the division instruction, and using this abstraction, we are able to accurately catch only the path going through lines 3 and 7. The second path that could terminate with a division by zero error goes through lines 5 and 7. The value for the denominator register, `r3` is unknown as the abstraction does not learn from the conditions of the two branch instructions. The unknown denominator produces two possible behaviors, for zero and non-zero values. The former yields an erroneous path, while the latter turns out to be the longest execution in the program.

$$\begin{array}{lll}
\text{X} & 1 \to 2 \to 3 \to 4 \to 6 \to 7 \to 8 \to 9 & (error\ path) \\
\surd & 1 \to 2 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 & (undetected\ infeasible\ 8\ t.u.) \\
\text{X} & 1 \to 2 \to 5 \to 6 \to 7 \to 8 \to 9 & (error\ path) \\
\surd & 1 \to 2 \to 3 \to 4 \to 6 \to 9 \to 10 & (7\ t.u.) \\
\surd & 1 \to 2 \to 5 \to 6 \to 9 \to 10 & (6\ t.u.)
\end{array}$$

The longest executable path in the program is an undetected infeasible path, introduced by the abstraction. Our goal now is to improve the initial simple abstraction to detect the erroneous path that goes through lines 5 and 7. We refine the first abstract semantics to update the values of the registers, when the program execution encounters a branch instruction, as we keep the state information unchanged, and modify only two semantic rules, for the `bne` and `beq` instructions, shown in Figure 7.3. We enhance new information learning, when one of the compared registers contains exact information that is used to update the other. Informally, the new operation, called `updReg` uses its first argument, the value of the program counter *PC*, to get the branch instruction and identify the registers. Using this abstraction, we are able to identify, for the example program, both erroneous paths.

$$\begin{array}{lll}
\text{X} & 1 \to 2 \to 3 \to 4 \to 6 \to 7 \to 8 \to 9 & (error\ path) \\
\text{X} & 1 \to 2 \to 5 \to 6 \to 7 \to 8 \to 9 & (error\ path) \\
\surd & 1 \to 2 \to 3 \to 4 \to 6 \to 9 \to 10 & (7\ t.u.) \\
\surd & 1 \to 2 \to 5 \to 6 \to 9 \to 10 & (6\ t.u.)
\end{array}$$

We work with the current implementation of the $\mathbb{K}$ framework, called $\mathbb{K}$-Maude. It is developed on top of Maude system and, in this way, it has access to all verification tools that Maude offers: a command for reachability analysis, an LTL model checker, an inductive theorem prover. We choose the first method, and perform reachability

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

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

Figure 7.3: Modified semantics rules for *SSRISC* branch instructions

analysis on our $\mathbb{K}$ specifications, to determine the WCET bounds. $\mathbb{K}$-Maude takes $\mathbb{K}$ specifications and generates rewrite theories in Maude. The two types of $\mathbb{K}$ rewrite rules, structural and computational are compiled into equations, and respectively rewrite rules.

A rewrite theory has an underlying equational theory, containing equations and membership statements, plus rewrite rules. A rewrite theory defines an abstract transitional system, where the equations represent, via equivalence classes, the states, while the rewrite rules represent the transitions between these classes. If the left-hand side of a rewrite rule matches the (fragment) of a current state, and the rule condition is satisfied, the system evolves into the state of the right-hand side of the particular rewrite rule. Maude system offers the possibility of unfolding this transition system and proving properties, or getting counterexample information.

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

7.2 Interval Analysis

The interval analysis [89] 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 present the interval analysis encoding on the integer domain. We approach this abstract interpretation based analysis from the perspective of keeping the formal executable semantics of *SSRISC* unchanged. To achieve this design and implementation desiderate, we need to manipulate the concrete definition of the system from Section 3.6 at a meta-level, taking advantage of the $\mathbb{K}$ framework configuration abstraction.

The interval analysis generalizes the constant propagation and we follow the same design steps as previously, from [115]. We start with the abstract domain, and in the case of the interval analysis, we need to define a new sort `IInt`, via a wrapping of two integer values standing for the lower and upper bound of an interval. For example, we represent the interval from 2 to 4 as $intv(2,4)$. This typing integration is seamless with respect to the semantics rules in $Language\ Semantics\ Module$, but the underlying *Builtin Module* should be modified. Also, the lattice reflects the partial order between intervals defined as follows: interval I_1 is "smaller" than interval I_2 if both the lower and upper bounds of I_1 are smaller than their corresponding bound in I_2.

The original *Builtin Module*, which provides the language (or support) operations for the *SSRISC* formal definition is now replaced by another which implements operations of intervals. For example, using our notation, the interval sum between $intv(2,4)$ and $intv(0,5)$ is $intv(2,9)$. The operations for multiplication and division are a bit more difficult, the division carrying also a division-by-zero exception. A multiplication of two intervals $intv(a_1,a_2)$ and $intv(b_2,b_2)$ is defined as follows:

$intv(a_1,a_2) \ \times \ intv(b_1,b_2) \ = \ intv(min(a_i \times b_i), max(a_i \times b_i))$ with $i = 1,2$

Once the abstract domain and the new *Builtin Module* are defined, we proceed to integrate the interval analysis, without modifying the semantics rules. In order to use the WRAP abstraction, that we present in Section 3.1, we identify the semantic entities that are necessary to define the interval analysis, in our case the integer register file `regs` and the special registers `lo` and `hi`. For simplification purposes, we leave aside the control-specific registers `pc` and `ra`.

The configuration for the interval analysis is the following:

$$\mathit{IAConfig} \equiv \langle\, \langle\, \mathsf{Map}[\mathit{Reg} \mapsto \mathit{IVal}]\, \rangle_{\mathsf{regs}}\, \langle \mathit{Val} \rangle_{\mathsf{lo}}\, \langle \mathit{Val} \rangle_{\mathsf{hi}}\, \rangle_{\mathsf{abst}}\, \langle \mathit{Val} \rangle_{\mathsf{mdop}}$$

The abstract configuration for the interval analysis consists of two sub-configurations, a language-based one with the important cells from the *SSRISC* language configuration and an abstraction-specific one. The three $\mathbb{K}$ cells, regs, lo and hi are wrapped and determine the language-specific part of the abstract configuration. The interval analysis requires another particular cell, called mdop because of the multiplication and division operations. We explain next why.

The *SSRISC* assembly language multiplication and division operations require two special registers hi and lo to deposit 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 a computed interval, which are produced and propagated during the interval-based operations. 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.

We discuss in detail, in Section 8.3 how a more complex example of an abstract-interpretation based analysis is defined over our concrete modular system (i.e. the hardware and software components).

7.3 Related Work

There are two immediate applications to the WCET analysis: automatic derivation of loop bounds and detection of infeasible paths. The necessity of using loop bounds is similar to the one for loop invariants in Hoare style program reasoning and therefore, tantamount to manual annotation. Loop bounds are important for the safety of the derived bounds. On the other hand, the infeasible paths detection helps to improve the accuracy of the timing analysis results.

We start with the field of automatically discovering of loop bounds. One successful approach is in the abstract execution method presented in [48]. The abstract execution is based on abstract interpretation and executes the programs with abstract values and abstract operations. It is different than abstract interpretation because abstract execution analyzes all the program executions in a separate way. The loop bounds are computed in the following way: the pair of bounds, initialized in the abstract domain are iterated through the loop iterations, incrementing a virtual execution counter. The join operation merges only the points with the same iteration count. The `aiT` tool use a combination of interval analysis and pattern matching to detect loop bounds [1]. A more recent work [67], that targets automatically loop bounds detection, uses deductive verification methods, with methods specifically designed for loop shapes. All these methods provide help and it is usually paired with manual annotation for tighter calculations.

The work on infeasible path detection is equally impressive. For example, the same method based on the abstract execution [48] is also used to detect infeasible pairs of nodes. Such a pair consists of two branching conditions that cannot be activated in the same context. The idea is to classify these pairs with respect to their behavior, by keeping information about the set of paths going through the two points in the pair. A different approach is in [52], where abstract data dependencies are collected and solved to flag a particular execution path as infeasible. These approaches as well as others rely on an underlying guarantee (assumptions or running tools like [28]) that the program is error free with respect to arithmetic errors. In this section we relax this guarantee and assume that the programs could contain errors like division by zero or arithmetic overflow. The paths that terminate in a error should be eliminated as the other infeasible path. In this regard, we use Maude's reachability analysis on the rewrite based theory of the *SSRISC* language definition and the program.

Chapter 8

A $\mathbb{K}$ Study of the ILP + AI Method

In this section we present the combined method of ILP + AI, popular in the WCET analysis community [108, 2], which enjoys an implementation applied on industrial scale applications. This implementation is called `aiT` [1]. There are two dedicated sections, one to each component of the ILP + AI component and a section on the implementation and experimentation using the $\mathbb{K}$-Maude tool. Before we describe in details each of these points, we present a general view on our $\mathbb{K}$ modeling of ILP + AI.

We consider that the two components represent abstractions applied either on the underlying program or the micro-architecture description of interest. The results of these abstract executions are the following:

- for the ILP part: a set of structural constraints that describe control flow information of the program.
- for the AI part: a classification of the program's instructions with respect to their cache memories

We design and implement these abstractions starting with the concrete representation of an embedded system, described in Section 3 (as an instance of a more generic modular system), in Section 4 (the software component represented by the formal executable semantics of *SSRISC*) and in Section 5 (the hardware component represented by the instruction and data cache models). Therefore, we propose an embedding of these abstractions directly over the concrete system, *without modifying it*. In comparison, the abstractions in Section 6 - the definitional unfolder with its subcomponent, the CFG extraction, require a slightly modified semantics (i.e. the rules on branch and jump instructions).

To achieve this design and implementation desiderate, we need to manipulate the concrete definition of the system from Section 3.6 at a meta-level. The 𝕂 framework implementation that we use, the 𝕂-Maude system support to little extent a meta-level handling of 𝕂 definitions (obviously, meta-level support is provided through the underlying Maude system). We approach this issue from the perspective of using one of the 𝕂's strengths, the configuration abstraction.

An abstraction simplifies the behavior of a concrete system achieving, in this ways the following two things: the problem size to be analyzed is reduced (in terms of the analyzed states) and the new reduced system preserves the properties of interest (with respect to the abstraction). For example, a problem of analyzing the data behavior of *SSRISC* programs requires both control features (i.e. the pc cell), information about the registers (i.e. the regs, fregs, hi, lo etc) and the main memory (and/or data caches). Therefore, we could assume that we select, from the entire configuration of a 𝕂 definition, only those 𝕂 cells that are necessary to compute abstract information. These cells, together with abstraction specific ones form the abstract configuration.

The goal of the ILP component is to transform the program into an integer linear program, representing execution path elements as linear constraints. These constraints are generated by traversing, symbolically, the entire program of interest. The abstract configuration contains, among other the pc cell, which records the program flow, from the *Language Semantics Module* and the cmem cell, which stores the program, from the *Main Memory Module*. The abstraction is graphically represented, at the level of modules, in Figure 3.7.

The AI component represents an abstract model of the micro-architecture behavior, in our case, of the cache memories. The results represent the categories of instructions that result in always hit, or always miss, in the cache. Therefore, apart from the abstraction specific cell, the abstract configuration wraps also the pc from the *Language Semantics Module*, the cmem from the *Main Memory Module* and the ic, ages, repl cells from the *IC Module*. Because of the parameterized nature of the concrete implementation, with this abstract configuration we define a family of abstractions, represented at the level of modules in Figure 3.8.

8.1 Abstractions for ILP Structural Constraints Extraction

We start with the subproblem of the path analysis, meaning how to generate constraints from the structure of the program, by executing it in a controlled (abstract) way. We present first in detail this algorithm - described in a 𝕂 module that we call *ILP Constraints*, then we explain implementation specific notions.

The ILP flow constraints for a given program are extracted through a breadth-first guided execution of the program semantics rules, as in [6]. A specialized 𝕂 cell, called kILP, manages this execution while it generates and collects the constraints in the constr cell. The configuration of this abstract semantics is the following:

$$ILPExConfig \equiv \langle Queue \rangle_{\mathsf{kILP}} \, \langle Idx \rangle_{\mathsf{loc}} \, \langle \mathsf{Map}[PC \mapsto (Low, Upp)] \rangle_{\mathsf{loopBnd}}$$

$$\langle \mathsf{Map}[PC \mapsto (List[InConstr], List[OutConstr])] \rangle_{\mathsf{constr}}$$

The kILP cell builds the working queue required for the breadth-first guided execution.

The loc cell holds a specialized counter for the index of the structural constraints. The cell loopBnd holds the manually added loop bounds, i.e. the functionality constraints, represented in the following way: the *PC* is the address of the backjump and the two integers *Low* and *Upp* represent the lower and the upper bounds on the number of loop iterations. The set of generated structural constraints is kept in the constr cell, and the two lists of variables cover the most general case of a program point with multiple incoming and outgoing edges. We omit the inclusion of a specialized cell to keep track of basic blocks and to reduce, in this way, the number of generated constraints. We briefly discuss such an extension at the end of this section.

The $\mathbb{K}$ rewrite rules, as shown in Fig. 8.1 for the subset language, are designed to simulate the breadth-first traversal of the control flow graph (CFG). A particularity of the approach is that we do not use an explicit representation of the CFG to extract the structural constraints. Instead, we symbolically execute the semantics rules while controlling the execution through the program counter value, *PC*. We choose the breadth-first strategy for illustration purposes on how to obtain the indexes in the loc cell. Any other traversal strategy would work as well. Before we explain the rules, we cover some general notations. The meta-notation x_{PC} represents the count variable associated with the instruction at the program point *PC*. *In* and *Out* are the lists of indexes for the incoming and outgoing edge variables, respectively. The notation *L In* represents that the constraint index *L* is added to the set *In*. For all the rules besides [FETCH], the outgoing edge constraint for the current instruction is also added as an incoming edge constraint for the next instruction.

The rule [SEQ] covers the case of a normal flow of execution, when the current instruction, represented by the pair $(PC : Instr)$, is executed. The *Queue* is also designed to maintain the previously visited program points and the `nq` operator inserts a new program point PC' in the *Queue*, provided that PC' is not labeled as visited. The [SPLIT] rule applies on the branch instructions. The execution is guided on both possible paths, as

RULE [SEQ]:

$$\langle \frac{PC{:}Instr\ Queue}{Queue\,\texttt{nq}\,(PC'{:}\cdot)} \rangle_{\mathsf{kILP}} \langle \frac{L}{L+1} \rangle_{\mathsf{loc}} \langle \cdots \frac{x_{PC}{\mapsto}(In,Out)\ x_{PC'}{\mapsto}(In',\ Out')}{x_{PC}{\mapsto}(In, L\,Out)\ x_{PC'}{\mapsto}(L\,In', Out')} \cdots \rangle_{\mathsf{constr}}$$

when $Instr =_{syntax}$ $\texttt{add/lw/sw}$ where $PC' = \texttt{next}(PC)$

RULE [SPLIT]:

$$\langle \frac{PC{:}(Mne\ \_,\_,Addr)\ Queue}{Queue\,\texttt{nq}\,(Addr{:}\cdot)\,\texttt{nq}\,(PC'{:}\cdot)} \rangle_{\mathsf{kILP}} \langle \frac{L}{L+2} \rangle_{\mathsf{loc}}$$

$$\langle \cdots \frac{x_{PC}{\mapsto}(In,Out)\ x_{PC'}{\mapsto}(In',Out')\ x_{Addr}{\mapsto}(In'',Out'')}{x_{PC}{\mapsto}(In, L\,L{+}1\,Out)\ x_{PC'}{\mapsto}(L{+}1\,In',Out')\ x_{Addr}{\mapsto}(L\,In'',Out'')} \cdots \rangle_{\mathsf{constr}}$$

when $Mne =_{syntax}$ $\texttt{bne/beq}$ where $PC' = \texttt{next}(PC)$

RULE [JMP]:

$$\langle \frac{(PC{:}(\texttt{j}\,Addr)\ Queue}{Queue\,\texttt{nq}\,(Addr{:}\cdot)} \rangle_{\mathsf{kILP}} \langle \frac{L}{L+1} \rangle_{\mathsf{loc}} \langle \cdots \frac{x_{PC}{\mapsto}(In,Out)\ x_{Addr}{\mapsto}(In',Out')}{x_{PC}{\mapsto}(In,L\,Out)\ x_{Addr}{\mapsto}(L\,In',Out')} \cdots \rangle_{\mathsf{constr}}$$

RULE [FETCH]: $\langle \cdots \frac{PC{:}\cdot}{PC{:}Instr} \cdots \rangle_{\mathsf{kILP}} \langle \cdots PC \mapsto Instr \cdots \rangle_{\mathsf{cmem}}$

Figure 8.1: 𝕂 Rules for Extracting ILP Structural Constraints

the two sequent instructions enqueue in the kILP cell. The set of constraints is modified, adding two outgoing edge constraints, with the indexes L and $L+1$, to the current program point PC and two incoming edge constraints, one for each of the branches, at the program points $\texttt{next}(PC)$ and $Addr$. The [JMP] rule is similar to the [SEQ] rule except that the subsequent program point at $\texttt{next}(PC)$ is replaced by the jump program point $Addr$. The [FETCH] rule brings the current instruction from the code memory cell, cmem. The set of constraints in constr cell is later combined with the set of constraints representing the loop bounds, in the loopBnd cell, and send to an external ILP solver.

The abstract semantics of ILP constraints generation produces the execution in Fig. 8.2 on the example program from Fig. 7.2. The constraints are represented as map elements from the instruction counter variable, x_i to pairs of incoming and outgoing lists of edge variables, d_i. One drawback of this approach consists of its inability to detect basic blocks and to characterize an entire block of instructions with a single counter variable. To accommodate on-the-fly basic block detection during the abstract execution, we propose an extension of the abstract configuration *ILPExConfig*. Next, we informally describe how this new optimization could be implemented, with respect to the rules in Fig. 8.1. We use a new cell to store an index for the current basic block that is modified according to the content of loc and the current instruction type. This new index value helps to distinguish between the cases when the instruction *Instr* starts a new basic block and when *Instr* is not the first instruction in a basic block. The rule [SEQ] splits to accommodate the previously mentioned two cases, while the rule [JMP] splits to distinguish between a forward jump and a backjump. The latter could potentially split an existing basic block into two basic blocks, conveniently updating the index in the loc cell.

The rewrite rules in Fig. 8.1 are presented at a meta-level, because of, for example, the meta-variables x_{PC} and x_{Addr} and the conditions of the [SEQ] and [SPLIT] rules which indicate decisions taken at the level of groups of instructions. We present in Section 8.3 how this abstraction for the ILP constraints generation is actually implemented and how the interplay between concrete and abstract execution steps are reflected at the $\mathbb{K}$ rules level, during the abstract execution.

8.2 Abstractions for Instruction Cache Behavior

The cache analyses from [112] use sets of concrete cache configurations to categorize memory references with respect to their behavior. There are three analyses - may, must and persistence - yielding the following four categories of memory references: always hit, always miss, persistent and not classified. For presentation purposes, we define, in a generic way, such analyses, starting from the previously introduced instruction cache modeling, in Section 5.2.

$$\langle 1 : \texttt{add r1, r2, r0}\rangle_{\mathsf{kIMP}}\ \langle 2\rangle_{\mathsf{loc}}\ \langle \cdots x_1 \mapsto (1,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{SEQ}}\ \langle 2 : \cdot\rangle_{\mathsf{kILP}}\ \langle 3\rangle_{\mathsf{loc}}\ \langle \cdots x_1 \mapsto (d_1,d_2)\ x_2 \mapsto (d_2,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{FETCH}}\ \langle 2 : \texttt{beq r2, r3, 10}\rangle_{\mathsf{kILP}}\ \langle 3\rangle_{\mathsf{loc}}\ \langle \cdots x_1 \mapsto (d_1,d_2)\ x_2 \mapsto (d_2,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{SPLIT}}\ \langle 10 : \cdot\ 3 : \cdot\rangle_{\mathsf{kILP}}\ \langle 5\rangle_{\mathsf{loc}}\ \langle \cdots x_2 \mapsto (d_2,d_3 d_4)\ x_{10} \mapsto (d_3,\cdot)\ x_3 \mapsto (d_4,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{2XFETCH}}\ \langle 10 : \texttt{sw r1, 4(r3)}\ 3 : \texttt{bne r1, r3, 6}\rangle_{\mathsf{kILP}}$$

$$\xrightarrow{\text{SEQ}}\ \langle 3 : \texttt{bne r1, r3, 6}\rangle_{\mathsf{kILP}}\ \langle 6\rangle_{\mathsf{loc}}\ \langle \cdots x_{10} \mapsto (d_3,d_5)\ \mathit{last} \mapsto (d_5,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{SPLIT}}\ \langle 6 : \cdot\ 4 : \cdot\rangle_{\mathsf{kILP}}\ \langle 8\rangle_{\mathsf{loc}}\ \langle \cdots x_3 \mapsto (d_4,d_6 d_7)\ x_6 \mapsto (d_6,\cdot)\ x_4 \mapsto (d_7,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{2XFETCH}}\ \langle 6 : \texttt{add r1, r3, r2}\ 4 : \texttt{sw r1, 4(r1)}\rangle_{\mathsf{kILP}}$$

$$\xrightarrow{\text{SEQ}}\ \langle 4 : \texttt{sw r1, 4(r1)}\ 7 : \cdot\rangle_{\mathsf{kILP}}\ \langle 9\rangle_{\mathsf{loc}}\ \langle \cdots x_6 \mapsto (d_6,d_8)\ x_7 \mapsto (d_8,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{SEQ}}\ \langle 7 : \cdot\ 5 : \cdot\rangle_{\mathsf{kILP}}\ \langle 10\rangle_{\mathsf{loc}}\ \langle \cdots x_4 \mapsto (d_7,d_8)\ x_5 \mapsto (d_9,\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{2XFETCH}}\ \langle 7 : \texttt{sw r2, 8(r1)}\ 5 : \texttt{j 8}\rangle_{\mathsf{kILP}}$$

$$\xrightarrow{\text{SEQ}}\ \langle 5 : \texttt{j 8}\ 8 : \cdot\rangle_{\mathsf{kILP}}\ \langle 11\rangle_{\mathsf{loc}}\ \langle \cdots x_7 \mapsto (d_8,d_{10})\ x_8 \mapsto (d_{10},\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{JMP}}\ \langle 8 : \cdot\rangle_{\mathsf{kILP}}\ \langle 12\rangle_{\mathsf{loc}}\ \langle \cdots x_5 \mapsto (d_9,d_{11})\ x_8 \mapsto (d_{10} d_{11},\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{FETCH}}\ \langle 8 : \texttt{add r2, r2, r3}\rangle_{\mathsf{kILP}}$$

$$\xrightarrow{\text{SEQ}}\ \langle 9 : \cdot\rangle_{\mathsf{kILP}}\ \langle 13\rangle_{\mathsf{loc}}\ \langle \cdots x_8 \mapsto (d_{10} d_{11},d_{12})\ x_9 \mapsto (d_{12},\cdot) \cdots\rangle_{\mathsf{constr}}$$

$$\xrightarrow{\text{FETCH}}\ \langle 9 : \texttt{j 2}\rangle_{\mathsf{kILP}}$$

$$\xrightarrow{\text{JMP}}\ \langle \cdot\rangle_{\mathsf{kILP}}\ \langle 14\rangle_{\mathsf{loc}}\ \langle \cdots x_9 \mapsto (d_{12},d_{13})\ x_2 \mapsto (d_2 d_{13},\ d_3 d_4) \cdots\rangle_{\mathsf{constr}}$$

Figure 8.2: Example of the Abstract Execution for ILP Constraints Generation

We proceed next to describe the may abstraction, shown in Fig. 8.3. There are two abstract instruction cache states, each represented at an organizational level in the figure and stored in a 𝕂 cell called ic. The first (partial) abstract cache has blocks 2 and 3 as the

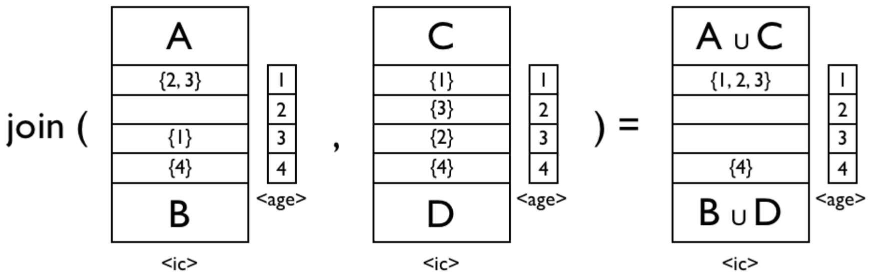


Figure 8.3: Join for the may analysis - union and minimal age

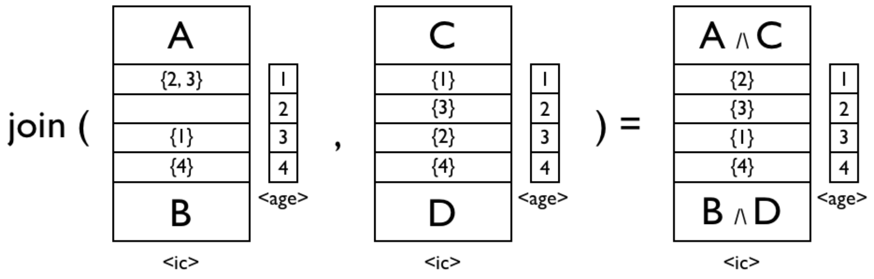


Figure 8.4: Join for the must analysis - intersection and maximal age

youngest ones in the cache, and block 4 the oldest. The second (partial) abstract cache has block 1 as the youngest and block 4 as the oldest.

We also need a cell age to encode the cache replacement policy, in this case the Least Recently Used (LRU) policy (as it is in [108]). An age value of 1 represents the youngest set of blocks. The set of ages is updated every time there is a cache read or write request. The middle part of each abstract cache state represents a particular cache block, with each line holding a set of memory blocks. For the abstract cache state in the righthand side, the middle part is the result of the union + minimal age of the corresponding information in the other two cache states. The join operation is applied on the entire cache content, the rest of a cache content is symbolically denoted with A to D. For the same abstract cache states, the join for the must analysis, shown in Fig. 8.4, relies on the following policy - intersection and maximal age.

Our goal is to reuse the support operations (i.e. cache replacement policies) from the concrete cache description. The abstract cache configuration *AICConfig* includes three

RULE [SEQ]:

$$\langle\langle \frac{PC : Instr}{\mathtt{next}(PC) : \cdot} \rangle_{\mathsf{ka}} \langle \frac{ACache}{ACache'} \rangle_{\mathsf{aic}} \rangle_{\mathsf{abs}}$$

$$\langle \cdots PC \mapsto \frac{ColCache}{\mathtt{join}(ACache', ColCache)} \cdots \rangle_{\mathsf{collect}}$$

when $\mathtt{join}(ACache', ColCache) \neq ColCache\ and_{Bool}\ Instr =_{syntax} \mathtt{add}/\mathtt{lw}/\mathtt{sw}$
where $ACache' = \mathtt{replace}(ACache, PC)$

RULE [JMP]:

$$\langle\langle \frac{PC : (\mathtt{j}\ Addr)}{Addr : \cdot} \rangle_{\mathsf{ka}} \langle \frac{ACache}{ACache'} \rangle_{\mathsf{aic}} \rangle_{\mathsf{abs}}$$

$$\langle \cdots PC \mapsto \frac{ColCache}{\mathtt{join}(ACache', ColCache)} \cdots \rangle_{\mathsf{collect}}$$

when $\mathtt{join}(ACache', ColCache) \neq ColCache$
where $ACache' = \mathtt{replace}(ACache, PC)$

Figure 8.5: $\mathbb{K}$-rules for instruction cache abstraction - SEQ and JMP

new cells, ka is the abstract k cell used to guide the execution, aic maintains the abstract instruction cache, and collect holds information about the current iteration through the abstract execution. Note that the cache behavior abstraction relies on concrete support operations like `compPl` to compute abstract cache states and to ensure coherence between the concrete and abstract cache behavior. This coherence check is obtained by subsorting the concrete cache to the abstract cache, and extending the specification of `compPl` to the abstract cache. This subsorting is possible in this particular case because the abstract cache contains a list of instructions, hence the *Instr* sort is subsorted to $\mathtt{List}[Instr]$.

$$AICConfig \equiv \langle \langle K \rangle_{\mathsf{ka}} \langle AICache \rangle_{\mathsf{aic}} \rangle_{\mathsf{abs*}} \langle \mathsf{Map}[PC \mapsto AICache] \rangle_{\mathsf{collect}}$$

where $AICache = \mathsf{Map}[Addr \mapsto \mathsf{List}[Instr]]$.

The $\mathbb{K}$ definition of the abstract cache analysis, in Fig. 8.6, uses rule schemas. A particularity of this encoding is that it relies on the concrete instruction cache and, RULE [SPLIT]:

$$\langle\langle \frac{PC : (Mne\, R_1, R_2, Addr)}{\texttt{next}(PC) : \cdot} \rangle_{\mathsf{ka}} \langle \frac{ACache}{ACache'} \rangle_{\mathsf{aic}} \rangle_{\mathsf{abs}}$$

$$\frac{\cdot}{\langle\langle Addr : \cdot\rangle_{\mathsf{ka}} \langle ACache'\rangle_{\mathsf{aic}}\rangle_{\mathsf{abs}}} \quad \langle \cdots PC \mapsto \frac{ColCache}{\texttt{join}(ACache', ColCache)} \cdots\rangle_{\mathsf{collect}}$$

when $\texttt{join}(ACache', ColCache) \neq ColCache\ and_{Bool}\ Mne =_{syntax} \texttt{bne/beq}$
where $ACache' = \texttt{replace}(ACache, PC)$

RULE [ELIM]:

$$\frac{\langle\langle PC : Instr\rangle_{\mathsf{ka}} \langle ACache\rangle_{\mathsf{aic}}\rangle_{\mathsf{abs}}}{\cdot} \quad \langle \cdots PC \mapsto \quad ColCache \quad \cdots\rangle_{\mathsf{collect}}$$

when $\texttt{join}(ACache', ColCache) = ColCache$
where $ACache' = \texttt{replace}(ACache, PC)$

Figure 8.6: 𝕂-rules for instruction cache abstraction - SPLIT and ELIM

because of the modularity of the design and the communication messages, it relies also on language definition and the main memory. Therefore, the ka cell uses the language executable semantics to go through all the program instructions and to update the corresponding abstract cache state, in the aic cell, and the global information, in the collect cell. An abstract execution that uses the rules in Fig. 8.6 is guided in the same breadth-first manner as the abstract execution for the ILP constraints extraction, with respect to the cell abs.

The rules [SEQ] and [JMP] update the abstract cache variable *ACache* according to the abstract replacement policy `replace` which extends the concrete replacement policy implemented with the `compPl` operation. Note that both these rules use the information in the cell collect and are applied only when the current instruction modifies the collected cache state *ColCache* (this is enforced by the rules conditions). When the collected cache variable is not modified by the abstract cache, the rule [ELIM] is applied. This rule voids the abs cell as the current program point does not add new information.

The [SPLIT] rule applies when the execution reaches a branch instruction, resulting in two abstract computations. This process creates a new abs cell to handle the jump-to

$$\langle \langle \underline{1 : \texttt{add r1, r2, r0}} \rangle_{\textsf{ka}} \, \langle 0 \mapsto [] \; 1 \mapsto [] \; 2 \mapsto [] \; 3 \mapsto [] \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \cdots 1 \mapsto [] \cdots \rangle_{\textsf{collect}}$$

$$\xrightarrow{\text{SEQ}}$$

$$\langle \langle \texttt{next}(1) : \cdot \rangle_{\textsf{ka}} \, \langle ACache' \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \cdots 1 \mapsto \texttt{join}(ACache', []) \cdots \rangle_{\textsf{collect}}$$
where $ACache' = \texttt{replace}(0 \mapsto [] \; 1 \mapsto [] \; 2 \mapsto [] \; 3 \mapsto [], 1)$

i.e., $ACache' = 0 \mapsto [1] \; 1 \mapsto [] \; 2 \mapsto [] \; 3 \mapsto []$, hence $\texttt{join}(ACache', []) \neq []$.

$$\langle \langle \underline{2 : \texttt{beq r2, r3, 10}} \rangle_{\textsf{ka}} \, \langle 0 \mapsto [1] \; 1 \mapsto [] \; 2 \mapsto [] \; 3 \mapsto [] \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \cdots 2 \mapsto [] \cdots \rangle_{\textsf{collect}}$$

$$\xrightarrow{\text{SPLIT}}$$

$$\langle \langle \texttt{next}(2) : \cdot \rangle_{\textsf{ka}} \, \langle ACache' \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \langle 10 : \cdot \rangle_{\textsf{ka}} \, \langle ACache' \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}}$$
$$\langle \cdots 2 \mapsto \texttt{join}(ACache', []) \cdots \rangle_{\textsf{collect}}$$
where $ACache' = \texttt{replace}(0 \mapsto [1] \; 1 \mapsto [] \; 2 \mapsto [] \; 3 \mapsto [], 2)$

i.e., $ACache' = 0 \mapsto [2] \; 1 \mapsto [1] \; 2 \mapsto [] \; 3 \mapsto []$, hence $\texttt{join}(ACache', []) \neq []$.

$$\langle \langle \underline{3 : \texttt{bne r1, r3, 6}} \rangle_{\textsf{ka}} \, \langle 0 \mapsto [2] \; 1 \mapsto [1] \; 2 \mapsto [] \; 3 \mapsto [] \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \cdots 3 \mapsto [] \cdots \rangle_{\textsf{collect}}$$

$$\xrightarrow{\text{SPLIT}}$$

$$\langle \langle \texttt{next}(3) : \cdot \rangle_{\textsf{ka}} \, \langle ACache' \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \langle 6 : \cdot \rangle_{\textsf{ka}} \, \langle ACache' \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}}$$
$$\langle \cdots 3 \mapsto \texttt{join}(ACache', []) \cdots \rangle_{\textsf{collect}}$$
where $ACache' = \texttt{replace}(0 \mapsto [2] \; 1 \mapsto [1] \; 2 \mapsto [] \; 3 \mapsto [], 3)$

i.e., $ACache' = 0 \mapsto [3] \; 1 \mapsto [2] \; 2 \mapsto [1] \; 3 \mapsto []$, hence $\texttt{join}(ACache', []) \neq []$.

$$\langle \langle \underline{4 : \texttt{sw r1, 4(r1)}} \rangle_{\textsf{ka}} \, \langle 0 \mapsto [3] \; 1 \mapsto [2] \; 2 \mapsto [1] \; 3 \mapsto [] \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \cdots 4 \mapsto [] \cdots \rangle_{\textsf{collect}}$$

$$\xrightarrow{\text{SEQ}}$$

$$\langle \langle \texttt{next}(4) : \cdot \rangle_{\textsf{ka}} \, \langle ACache' \rangle_{\textsf{aic}} \, \rangle_{\textsf{abs}} \, \langle \cdots 4 \mapsto \texttt{join}(ACache', []) \cdots \rangle_{\textsf{collect}}$$
where $ACache' = \texttt{replace}(0 \mapsto [3] \; 1 \mapsto [2] \; 2 \mapsto [1] \; 3 \mapsto [], 4)$

i.e., $ACache' = 0 \mapsto [4] \; 1 \mapsto [3] \; 2 \mapsto [2] \; 3 \mapsto [1]$, hence $\texttt{join}(ACache', []) \neq []$.

Figure 8.7: Example of the Abstract Execution for Instruction Cache Behavior (part 1)

computation, while the current abs cell maintains the fall-through computation. Note that there is also a similar [FETCH] rule, as in the ILP abstraction (Fig. 8.1), not shown here.

We present, in Fig. 8.7 and continue in Fig. 8.8, how the $\mathbb{K}$-rules for the instruction cache abstraction apply on the path [1,2,3,4,5,8,9] of the example program in Fig. 7.2. The abstract execution considers a fully associative cache, with 4 cache lines and an

$\langle\langle \underline{5 : \mathtt{j\,8}}\rangle_{\mathsf{ka}}\ \langle 0 \mapsto [4]\ 1 \mapsto [3]\ 2 \mapsto [2]\ 3 \mapsto [1]\rangle_{\mathsf{aic}}\ \rangle_{\mathsf{abs}}\ \langle \cdots 5 \mapsto [\,] \cdots\rangle_{\mathsf{collect}}$

$\overset{\text{JMP}}{\to}$

$\langle\langle 8 : \cdot\rangle_{\mathsf{ka}}\ \langle \mathit{ACache}'\rangle_{\mathsf{aic}}\ \rangle_{\mathsf{abs}}\ \langle \cdots 5 \mapsto \mathtt{join}(\mathit{ACache}', [\,]) \cdots\rangle_{\mathsf{collect}}$
where $\mathit{ACache}' = \mathtt{replace}(0 \mapsto [3]\ 1 \mapsto [2]\ 2 \mapsto [1]\ 3 \mapsto [\,], 4)$

i.e., $\mathit{ACache}' = 0 \mapsto [5]\ 1 \mapsto [4]\ 2 \mapsto [3]\ 3 \mapsto [2]$, hence $\mathtt{join}(\mathit{ACache}', [\,]) \neq [\,]$.
... after the execution of the other abs cell $\langle \cdots \langle 6 : \cdot\rangle_{\mathsf{ka}} \cdots\rangle_{\mathsf{abs}}$,
the collecting cell contains $\langle \cdots 8 \mapsto (0 \mapsto [8]\, 1 \mapsto [7]\, 2 \mapsto [6]\, 3 \mapsto [3]) \cdots\rangle_{\mathsf{collect}}$
$\langle\langle \underline{8 : \mathtt{addr2,\ r2,\ r3}}\rangle_{\mathsf{ka}}\ \langle 0 \mapsto [5]\ 1 \mapsto [4]\ 2 \mapsto [3]\ 3 \mapsto [2]\rangle_{\mathsf{aic}}\ \rangle_{\mathsf{abs}}$
$\langle \cdots 8 \mapsto (0 \mapsto [8]\, 1 \mapsto [7]\, 2 \mapsto [6]\, 3 \mapsto [3]) \cdots\rangle_{\mathsf{collect}}$

$\overset{\text{SEQ}}{\to}$

$\langle\langle \mathtt{next}(8) : \cdot\rangle_{\mathsf{ka}}\ \langle \mathit{ACache}'\rangle_{\mathsf{aic}}\ \rangle_{\mathsf{abs}}$
$\langle \cdots 8 \mapsto \mathtt{join}(\mathit{ACache}', (0 \mapsto [8]\, 1 \mapsto [7]\, 2 \mapsto [6]\, 3 \mapsto [3])) \cdots\rangle_{\mathsf{collect}}$
where $\mathit{ACache}' = \mathtt{replace}(0 \mapsto [5]\ 1 \mapsto [4]\ 2 \mapsto [3]\ 3 \mapsto [2], 8)$

i.e., $\mathit{ACache}' = 0 \mapsto [8]\ 1 \mapsto [5]\ 2 \mapsto [4]\ 3 \mapsto [3]$,
hence $\mathtt{join}(\mathit{ACache}', (0 \mapsto [8]\, 1 \mapsto [7]\, 2 \mapsto [6]\, 3 \mapsto [3])) =$
$= (0 \mapsto [8]\, 1 \mapsto [7,5]\, 2 \mapsto [6,4]\, 3 \mapsto [3]) \neq (0 \mapsto [8]\, 1 \mapsto [7]\, 2 \mapsto [6]\, 3 \mapsto [3])$.
$\langle\langle \underline{9 : \mathtt{j\,2}}\rangle_{\mathsf{ka}}\ \langle 0 \mapsto [8]\ 1 \mapsto [7,5]\ 2 \mapsto [6,4]\ 3 \mapsto [3]\rangle_{\mathsf{aic}}\ \rangle_{\mathsf{abs}}\ \langle \cdots 9 \mapsto [\,] \cdots\rangle_{\mathsf{collect}}$

$\overset{\text{JMP}}{\to}$

$\langle\langle 2 : \cdot\rangle_{\mathsf{ka}}\ \langle \mathit{ACache}'\rangle_{\mathsf{aic}}\ \rangle_{\mathsf{abs}}\ \langle \cdots 9 \mapsto \mathtt{join}(\mathit{ACache}', [\,]) \cdots\rangle_{\mathsf{collect}}$
where $\mathit{ACache}' = \mathtt{replace}(0 \mapsto [8]\ 1 \mapsto [7,5]\ 2 \mapsto [6,4]\ 3 \mapsto [3], 9)$

i.e., $\mathit{ACache}' = 0 \mapsto [9]\ 1 \mapsto [8]\ 2 \mapsto [7,5]\ 3 \mapsto [6,4]$, hence $\mathtt{join}(\mathit{ACache}', [\,]) \neq [\,]$.
...

Figure 8.8: Example of the Abstract Execution for Instruction Cache Behavior (part 2)

empty initial cache state. We work with the definition of `join` for the *may* analysis: union + maximal age [2]. For brevity, we do not emphasize the maximal age update from the age cell. (Note that when we want to switch to a *must* analysis, the only change in the instruction cache abstraction is that the `join` implements intersection + minimal age [2].) The two applications of the [SPLIT] rule, for instructions at program points 2 and 3, produce sequent execution threads, i.e. two abs cells. For presentation purposes, we choose to follow only the results of the fall-through cases. We also consider a computed abstract cache content at program point 6 and we use this particular information when the abstract execution reaches program point 8. The abstract computation reuses support operations defined in the concrete description of the instruction cache behavior. For example, the operation `replace` - the replacement policy for abstract cache states is used at every step of the execution and relies on the concrete counterpart, `compPl`.

Next, we explain that, when the execution applies rule [ELIM], i.e., the computation reaches the fixed point. After one unfolding, we get back to the instruction at program point 2. The current $ACache'$ is $0 \mapsto [9]$ $1 \mapsto [8]$ $2 \mapsto [7,5]$ $3 \mapsto [6,4]$, which joined with the current value of $ColCache$ (i.e., $0 \mapsto [1]$ $1 \mapsto []$ $2 \mapsto []$ $3 \mapsto []$) results into an update of the cell collect as follows:

$$\langle \cdots 2 \mapsto (0 \mapsto [9,1]\ 1 \mapsto [8]\ 2 \mapsto [7,5]\ 3 \mapsto [6,4]) \cdots \rangle_{\mathsf{collect}}$$

After another unfolding, which follows the same steps as in Fig. 8.7, the abstract execution reaches again the instruction at program counter 2 with the same $ACache'$. Now, $\texttt{join}(ACache', ColCache) = ColCache$. At this point, the rule [ELIM] is applied instead of the rule [SPLIT] and the abstract execution stops. It is important to notice that this abstract execution is guided in the same breadth-first manner as the abstract execution for the ILP constraints extraction, wrt the cell abs.

8.3 Implementation

In this section we present how to integrate the ILP structural constraints generation and the AI-based abstraction for cache analysis over our $\mathbb{K}$ specification of a modular architecture, described in Section 3. We follow a configuration-based abstraction methodology, meaning we identify the necessary elements from the concrete system configuration, we wrap them and integrate in a new, abstract configuration. We use

$\mathbb{K}$-Maude [29], a prototype implementation of the $\mathbb{K}$ framework done on top of Maude system [24]. We present first how the abstract configuration are build and then how the interleaving between concrete and abstract executions work.

The abstraction specific configuration of the ILP structural constraints extraction is the following:

$$\langle\langle K\rangle_{\mathsf{k}}\langle Addr\rangle_{\mathsf{getAddr}}\langle Idx\rangle_{\mathsf{ctridx}}\langle Map[PC \mapsto K]\rangle_{\mathsf{sconstr}}\rangle_{\mathsf{ilp}}$$

Apart from the (abstract) computational cell k, which in the meta configuration appears as kILP, the abstraction uses a storage cell called sconstr to collect, for each program point *PC*, all the generated constraints. The index *Idx* of the currently generated constraint is in the ctridx cell, and the target address, in case of a jump/branch instruction is in the getAddr. We use the $\mathbb{K}$ cell ilp to wrap the abstraction specific elements, there is also a sub-configuration derived from the concrete semantics which is not show here, but appears in a subsequent rewrite rule. This sub-configuration contains the concrete computational cell and the pc cell, wrapped under a new cell called concrete.

Similarly, the configuration of the AI-based analysis for instruction cache behavior is the following:

$$\langle\langle\langle K\rangle_{\mathsf{k}}\langle\langle Map\rangle_{\mathsf{ic}}\langle Map\rangle_{\mathsf{ages}}\rangle_{\mathsf{aic}}\rangle_{\mathsf{abst}}\langle K\rangle_{\mathsf{absType}}$$

$$\langle Map[PC \mapsto K]\rangle_{\mathsf{collect}}\langle K\rangle_{\mathsf{joinRes}}$$

$$\langle Addr\rangle_{\mathsf{fstAddr}}\langle Val\rangle_{\mathsf{instrLen}}\langle Map[K \mapsto Val]\rangle_{\mathsf{params}}\rangle_{\mathsf{ica}}$$

The AI-based analysis for the instruction cache behavior computes sets of abstract cache states and therefore requires a wrapping, in the aic cell, of the instruction cache module, with the cache content represented by the ic cell and the ages, represented by ages. The parameterization of the analysis (may, must or persistent) is represented in absType, while the architecture-related parameters are: in instrLen - the instruction length of the target language and in params - the cache parameters (i.e. size, associativity etc). The cells collect and joinRes hold the collected results during the abstract computation, respectively the computed result of in a join point. The cells aic absType, collect, joinRes are abstraction specific, while fstAddr, params etc are selected from the concrete configuration of the system.

We refer next to the computational part of the two abstractions, meaning how the abstract results are obtained from an interleaved execution between concrete and abstract parts. Our original system is modular and these modules communicate between them via communication messages. For example, the module *IC Module* sends the message `imiss` to the *Main Memory Module* and receives back the instruction `Ins`. Another example, in the case of a dummy cache memory, the *Language Semantics Module* issues the instruction fetch request to the *Main Memory Module* via `geti` message.

The algorithm for the ILP abstraction monitors the concrete execution between the language semantics and the main memory module, and when a message of interest is on top of the k cell, the flow transfers to the abstract counterpart to generate constraints (i.e. assign a constraint id, determine what are the constraints for the various program points involved). For example, the abstract execution starts with an abstract state, where the set of constraints is empty, the initial constraint index is zero and the first instruction is requested from the main memory (here we assume that the *IC Module* is not included, but the abstraction works equally well if it would be).

For example, the branch rewrite rule [SPLIT], presented in Fig. 8.1, is implemented in 𝕂-Maude, in a new module called *ILP Abstr Module* as in Fig. 8.9.

We present how the abstraction for structural ILP constraint extraction is mapped over the concrete specification of the system, in this particular rule [ILP-BINSTR], over the formal executable semantics of the language. This 𝕂 rule corresponds to the case of a branch instruction *Ins*, as guarded by the condition introduced with the keyword "when". The branch instruction *Ins* is at program point *PC*, the fall-through instruction

```
rule  [ILP-BINSTR] :
   (<wrp> <concrete>...
       <k> Ins:Instr ...</k>
       <pc> NPC:#Int32 </pc>
   ...</concrete> => . </wrp>)
   <ilp>...
      <k> (busy-waiting(PC) => (Addr :: null ~> NPC :: null))
       ...</k>
      <getAddr> Addr:#Int32 => . </getAddr>
      <ctridx> Idx:#Int32 => Idx +Int32 2 </ctridx>
      <sconstr>...
         (   PC |-> ctrPC (inPC(InsPC:CL), outPC(OutsPC:CL))
            NPC |-> ctrPC (inPC(InsNPC:CL), outPC(OutsNPC:CL))
           Addr |-> ctrPC (inPC(InsA:CL), outPC(OutsA:CL))
         ) =>
         (  PC |-> ctrPC(inPC(InsPC),
                    outPC(Idx, (Idx +Int32 1), OutsPC))
           NPC |-> ctrPC (inPC((Idx +Int32 1), InsNPC),
                    outPC(OutsNPC))
          Addr |-> ctrPC(inPC(Idx,InsA), outPC(OutsA))
          )
    ...</sconstr>
  ...</ilp>
when isBInstr(Ins)
```

Figure 8.9: The 𝕂 rewrite rule for branch in the ILP constraints generation

at *NPC*, and the instruction at the target address is at *Addr*.

The rule [ILP-BINSTR] has two parts, the concrete execution takes place in the wrp cell, while the abstract execution in the ilp cell. The concrete part of the execution has on top of the computation cell the current instruction, and because this particular ILP abstraction does not rely on data analysis, the actual execution of *Ins* is not necessary. Therefore, the entire concrete execution is reduced to ·, via local rewriting. The abstract part of the rewrite rule should collect generated constraints for all the program points of interest, with respect to the current instruction. A special token `busy-waiting` controls the abstract execution, which, in turn, is reduced to a list of program points that needs to be explored next.

This rule yields two constraints, each corresponding to the then and else branches of the instruction. Hence, to capture this flow information, it is necessary to generate two new constraint indexes, using the current index in the cell ctridx content. We collect, for all the program points, all the ILP structural constraints using in the cell sconstr. Therefore, each program point holds two lists of constraint indexes - `inPC` and `outPC` for input and respectively output flow counter. The `ctrPC` is just a wrapper for these two lists. For the current program point having the branch instruction, *PC*, the set of output constraints, `outsPC`, gets two new constraints with the indexes Idx and $Idx+1$. The same two constraints are also input constraints for the next possible program counters *Addr* and *NPC*. The index Idx is added to the input list of indexes *InsA* for *Addr* and the index $Idx+1$ to the input list of indexes *InsNPC* for *NPC*.

We discuss next the implementation of the AI-part of the ILP + AI method for WCET analysis. The may and must analyses share the same core of rewrite rules, parameterized by the abstraction type. In this case, the abstract execution monitors the communication between the *Language Semantics Module*, the *IC Module* and the *Main Memory Module*.

Next, we explain how we implement the cache behavior prediction in the 𝕂 framework. The key element is to identify the flow of the concrete execution, what messages are involved between language semantics, caches and main memory modules. Then, the abstract execution module, via its rewrite-rules, stops the concrete execution and process the data. In Figure 8.11 we present the 𝕂 rewrite rule for a generic join of abstract cache states, with an implicit representation of the control flow graph (CFG). Again, this rule has two parts - the wrapped concrete configuration in the wrp cell and the abstraction in the ica cell.

The key for this abstraction encoding is the sub-configuration of the instruction cache content, stored in the ic cell and the respective ages in the `ages` cell. We denote this sub-configuration as ic + ages. The actual join operation is implemented as an external

```
rule [AI-JOIN] :
  <ica>...
    <abst>
     <k> busy-waiting(PC:#Int32) ~> (NPC:#Int32 :: null) ...</k>
     <aic> <ic> C:Map => C1 </ic> <ages> A:Map => A1 </ages> </aic>
    </abst>
    <absType> AbsType:K </absType>
    <collect>... PC |-> ColCache:K ...</collect>
    <joinRes> . => join(
            <ic> C </ic> <ages> A </ages>,
             <ic> C1 </ic> <ages> A1 </ages>,
             ColCache, AbsType,
             getCLinePC(wia(FAddr),FAddr,ILen,NL,M),M, NL)
      </joinRes>
      <fstaddr> FAddr:#Int32 </fstaddr>
      <instrlen> ILen:#Int32 </instrlen>
      <params>... csize |-> NL:#Int32 ca |-> M:#Int32 ...</params>
 ...</ica>
 <wrp> <concrete>...
    <ic> C1:Map </ic> <ages> A1:Map </ages>
    ...</concrete> => .Bag
 </wrp>
```

Figure 8.10: The 𝕂 rewrite rule for parametric join in the cache behavior analysis

function, called `join`, whose result populates the joinRes cell. Two of the `join` function arguments are the sub-configurations ic + ages in the wrapped concrete cell wrp and in the aic cell in the abstract part, abst.

The [AI-JOIN] rewrite rule, in Fig. 8.10 shows several degrees of parameterization of our abstraction encoding. First, we have a special cell absType that contains the abstract type (may, must or persistent). Second, the cache parameters are stored in the params, with `csize` and `ca` representing the cache size, and respectively the cache associativity. Third, there is also a program-dependent parameter, in the fstAddr cell that represents the address of its first instruction. This helps to map to the instruction cache, the code stored in the main memory. Forth, a parameter for the size of one instruction, in the instrLen cell.

```
rule [AI-JOINCLT] :
  join( _, <ic> C1:Map </ic> <ages> A1:Map </ages>,
           wbagk(<ic> C:Map </ic> <ages> A:Map </ages>),
           may, gotCLine(L:List), M:#Int32, NL:#Int32 )
     =>
  joinF(<ic> cofilter(C1,L) </ic> <ages> cofilter(A1,L) </ages>,
        <ic> cofilter(C,L) </ic>  <ages> cofilter(A,L) </ages>,
        wbagk(
         <ic>
           joinBlock(
            <ic> filter(C,L) </ic> <ages> filter(A,L) </ages>,
            wbagk (
            <ic> filter(C1,L) </ic> <ages> filter(A1,L) </ages>),
            may, L, .Set, 1)
          </ic>
          <ages> filter(A1,L) </ages>
          ), may, L, M, NL )
```

Figure 8.11: Join function (partial) for the may analysis

Typical to an abstract interpretation based computation, we collect, in the collect cell, for each program point *PC*, a set of abstract cache states, *ColCache* (the sub-configurations $\mathsf{ic}+\mathsf{ages}$). All the parameters mentioned above are necessary to implement the `join` operation, for the may analysis, in Figure 8.11. The function follows the organization of the instruction caches, using several auxiliary functions and operations. For example, the complementary `filter` and `cofilter` functions slice the instruction cache, together with the corresponding age, based on a previously computed list *L* of cache addresses. The list *L* is calculated using the cache size and associativity, for the current program point *PC*. The actual join operation is performed block by block (using `joinBlock` function), for the instruction cache content (in the ic cell).

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. Each of the abstractions, described in 𝕂 modules, are compiled, using 𝕂-Maude tool [29] into Maude modules, each being a rewrite theory. During our abstract computation (either for the ILP or AI part), the 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 𝕂-Maude. When we measure the percentage of the abstract execution steps with respect to the total number of execution steps, we eliminate from this counting, the tool generated ones. Hence, the total number of rewrites, which appear under the column **total rewrites** in all three tables in Figures 8.12, 8.13 and 8.14, consists only of the rewrite rules and equations of our system's specification.

We select several of the Mälardalen benchmark programs [47] 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 Figure 8.12.

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 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 simple (we recall that only the pc register and the code memory are necessary to detect the program's flow). Also, this analysis does not rely on abstract external functions, such as the join operation in the AI counterpart, on whose results we discuss next.

The tables in Figure 8.13 and Figure 8.14 display the experimental results on the abstract interpretation-based analyses, may and respectively must, for the instruction cache behavior. Each table contains the results of the analysis with an implicit control flow graph (CFG). Running both may and must analyses, having an implicit representation of the CFG yields the following results in terms of reusability, the percentages

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

Figure 8.12: Results for the ILP structural constraints extraction

name	**concrete** (%) (impl CFG)	**abstract** (%) (impl CFG)	**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

Figure 8.13: Results for the AI-based (may) analysis for instruction cache behavior with implicit CFG

name	**concrete** (%) (impl CFG)	**abstract** (%) (impl CFG)	**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

Figure 8.14: Results for the AI-based (must) analysis for instruction cache behavior with implicit CFG

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).

The instructions are classified, according to these abstractions, into always hit, always miss, first miss (persistent). Our implementation addresses this issue in a support tool to our 𝕂 definition. We process the output configuration, after the analyses are finished, and we extract the structural ILP constraints, respectively the abstract cache states corresponding to each program point. The former constraints are dumped into a specification file for an external ILP solver, together with loop bounds or other manually given ILP functional constraints. This ILP formulation is refined with the results from the three analyses and a new specification file is generated. We use the solver called `lp_solve` and presented in [10]. `lp_solve` is implemented in C and is a non-comercial linear programming solver, capable of solving a number of instances of linear programming (i.e. pure linear, integer linear, mixed integer, 0-1 etc). We have integrated this in our 𝕂 specification.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

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

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

We designed modular and parametric micro-architecture (i.e. instruction and data cache memories), to facilitate reasoning about the execution of an embedded program on a specified machine. Then, we investigate, with the direct application in WCET analysis a well-known combined methodology - ILP+AI. The implementation took advantage of the following two key elements: the underlying technology of context transformation, provided by the $\mathbb{K}$ framework and our proposed embedding to manipulate abstract and concrete configurations. We tested the current implementation on several benchmark programs, with the formal semantics kept unmodified, measuring a factor of reusability of trusted, concrete definition.

In the context of a new methodology for WCET analysis of hard-real time programs, that is centered around an assembly language definition, we also presented two abstractions for CFG extraction and unfolding. First, we extended the formal executable semantics of the language with control information and apply reachability techniques, via Maude `search` command, to extract an over-approximation of the CFG. Then, based on this result and together with manual loop bounds annotations, we defined a semantic program unfolder that was used to compute the set of all possible program executions. The method assumed a number of restrictions in terms of annotations, as well as of structure of the CFG.

With respect to the existing WCET estimation approaches, this is the first use of reachability analysis, on the formal executable semantics of an assembly language to extract control flow information and to provide the set of all possible program executions in the presence of various constraints. With respect to rewriting logic in general and $\mathbb{K}$ in particular, this is the first use of rewriting-based techniques to extract program flow information.

9.2 Future Work

In the strict sense of the definition of the *SSRISC* language, this current work should enjoy a twofold extension. First, we plan to integrate the IO system of [3] to use a third party implementation of support operations for various arithmetic structures. Second, the main memory modeling is more complex than what we exemplify with in Section 4. For example, the actual MIPS [14] data memory has a data segment where the program data is, a heap for extra space and a stack to handle subroutine calls. In our modeling we do not formally distinguish between these. Our goal is to use the formal executable semantics for timing analysis of embedded software systems.

The obvious improvement of our semantics-based WCET analysis is to derive more powerful abstract semantics and to eliminate not only error paths, but much of the infeasible execution paths. We place equal importance on micro-architecture modeling for processor behavior analysis and, in short term, we plan to to integrate $\mathbb{K}$-based pipeline models. These steps converge towards the first WCET analyzer using the rewrite-based technology. Also as future work, we propose to define a data analysis to compute bounds for program variables and to use them to improve the approximation of the CFG. With respect to the program unfolder, we propose to improve the assertion language to introduce further control-flow related constraints.

Our future plans include the replacement of the ad-hoc definitions of the instruction and data cache behavior with their respective implementation in a hardware description language (HDL). Then the corresponding abstractions monitor a formal executable semantics of the HDL, say for example Verilog [82]. Also, the recent advances in deductive verification methods in rewriting logic, the matching logic approach [102] present a promising alternative to a deductive-style WCET analyzer. We plan to integrate both the *SSRISC* language definition and the underlying cache memory models, as Verilog programs, to formally verify timing properties of programs running in a critical system.

References

[1] AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, pages 52–66. Springer, 1996.

[3] A. Arusoaie, T. F. Şerbănuţă, C. Ellison, and G. Roşu. Making Maude definitions more interactive. *Rewriting Logic and Its Applications, WRLA 2012*, volume 7571 of *Lecture Notes in Computer Science*. Springer, 2012.

[4] M. Asavoae. K semantics for assembly languages : A case study. *K11 Workshop*, 2011. submitted.

[5] M. Asavoae and I. M. Asavoae. Using the executable semantics for cfg extraction and unfolding. *SYNASC*, pages 123–127, 2011.

[6] M. Asavoae, I. M. Asavoae, and D. Lucanu. On abstractions for timing analysis in the k framework. *FOPARA*, volume 7177 of *LNCS*, pages 90–107, 2011.

[7] M. Asăvoae, D. Lucanu, and G. Roşu. Towards semantics-based WCET analysis. *WCET*, 2011.

[8] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. *CC*, pages 5–23, 2004.

[9] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Developing UPPAAL over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.

[10] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 1 2004. Available at <http://lpsolve.sourceforge.net/5.5/>. Last accessed Dec, 18 2009.

[11] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. volume 36, pages 1–8, 2011.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[13] F. Bodin and I. Puaut. A wcet-oriented static branch prediction scheme for real time systems. *ECRTS*, pages 33–40, 2005.

[14] R. Britton. *MIPS Assembly Language Programming*. Pearson Education, 2003.

[15] B. Brock and W. A. H. Jr. Formally specifying and mechanically verifying programs for the motorola complex arithmetic processor dsp. *ICCD*, pages 31–36, 1997.

[16] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10\hat{2}0$ states and beyond. *LICS*, pages 428–439, 1990.

[18] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25:13–25, June 1997.

[19] B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 147–163, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. *ESEC/SIGSOFT FSE*, pages 5–14, 2007.

[21] V. Chvatal. *Linear Programming*. Freeman, 1993.

[22] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. *Logic of Programs*, pages 52–71, 1981.

[23] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in maude. *Electr. Notes Theor. Comput. Sci.*, 15:331–352, 1998.

[24] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[25] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. *CODES+ISSS*, pages 267–276, 2010.

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

[27] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[28] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, 2005.

[29] T. F. Şerbanuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. *WRLA 2010*, volume 6381 of *LNCS*, pages 104–122, 2010.

[30] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM, pages 301–310, Washington, DC, USA, 2005. IEEE Computer Society.

[31] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular execution time analysis using model checking. *WCET*, pages 113–123, 2010.

[32] N. Dave, M. Katelman, M. King, Arvind, and J. Meseguer. Verification of microarchitectural refinements in rule-based systems. *MEMOCODE*, pages 61–71, 2011.

[33] F. Durán and J. Meseguer. Parameterized theories and views in full maude 2.0. *Electr. Notes Theor. Comput. Sci.*, 36:316–338, 2000.

[34] F. Durán and J. Meseguer. A church-rosser checker tool for conditional order-sorted equational maude specifications. *WRLA*, pages 69–85, 2010.

[35] F. Durán and J. Meseguer. A maude coherence checker tool for conditional order-sorted rewrite theories. *WRLA*, pages 86–103, 2010.

[36] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker and its implementation. *SPIN*, pages 230–234, 2003.

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

[38] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *TACAS*, pages 87–106, 1996.

[39] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of java programs in javafan. *CAV*, pages 501–505, 2004.

[40] A. Farzan, J. Meseguer, and G. Rosu. Formal jvm code analysis in javafan. *AMAST*, pages 132–147, 2004.

[41] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[42] A. C. J. Fox. Formal specification and verification of arm6. *TPHOLs*, pages 25–40, 2003.

[43] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. *ITP*, pages 243–258, 2010.

[44] M. K. Ganai, A. Gupta, and P. Ashar. Efficient modeling of embedded memories in bounded model checking. *CAV*, pages 440–452, 2004.

[45] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[46] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. *CAV*, pages 72–83, 1997.

[47] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen wcet benchmarks: Past, present and future. *WCET*, pages 136–146, 2010.

[48] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. *RTSS*, pages 57–66, 2006.

[49] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. *IEEE Real-Time Systems Symposium*, pages 456–466, 2008.

[50] D. Hardy and I. Puaut. Wcet analysis of instruction cache hierarchies. *Journal of Systems Architecture - Embedded Systems Design*, 57(7):677–694, 2011.

[51] N. A. Harman. Verifying a simple pipelined microprocessor using Maude. *WADT '01*, pages 128–151. Springer-Verlag, 2001.

[52] C. A. Healy and D. B. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. *IEEE Real Time Technology and Applications Symposium*, pages 79–88, 1999.

[53] M. Hills. Memory representations in rewriting logic semantics definitions. *Electr. Notes Theor. Comput. Sci.*, 238(3):155–172, 2009.

[54] M. Hills and G. Rosu. Towards a module system for k. *WADT*, pages 187–205, 2008.

[55] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. *VMCAI*, pages 78–78, 2005.

[56] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[57] M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, 2010.

[58] I. Kadayif, M. T. Kandemir, N. Vijaykrishnan, and M. J. Irwin. An integer linear programming-based tool for wireless sensor networks. *J. Parallel Distrib. Comput.*, 65(3), 2005.

[59] D. Kästner and M. Langenbach. Code optimization by integer linear programming. *CC*, pages 122–136, 1999.

[60] D. Kästner and S. Wilhelm. Generic control flow reconstruction from assembly code. *LCTES*, pages 46–55. ACM, 2002.

[61] M. Katelman, S. Keller, and J. Meseguer. Concurrent rewriting semantics and analysis of asynchronous digital circuits. *WRLA*, pages 140–156, 2010.

[62] M. Katelman and J. Meseguer. A rewriting semantics for abel with applications to hardware/software co-design and analysis. *Electr. Notes Theor. Comput. Sci.*, 176(4):47–60, 2007.

[63] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[64] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Inf. Comput.*, 163(1):203–243, 2000.

[65] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. *VMCAI*, pages 214–228, 2009.

[66] R. Kirner and P. Puschner. Time-predictable computing. *Proc. 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, Oct. 2010.

[67] J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for wcet analysis. *Ershov Memorial Conference*, pages 227–242, 2011.

[68] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao. Partitioning and pipelined scheduling of embedded system using integer linear programming. *Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops - Volume 02*, ICPADS '05, pages 37–41, Washington, DC, USA, 2005. IEEE Computer Society.

[69] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. *SAS*, pages 294–309, 2002.

[70] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. *Static Analysis Symposium*, pages 294–309, 2002.

[71] X. Leroy. Formal verification of a realistic compiler. *Communications of ACM*, 52(7):107–115, 2009.

[72] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Journal of Real-Time Systems*, 29:2005, 2005.

[73] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Journal Of Real-Time Systems*, 34:2006, 2005.

[74] X. Li, L. Yun, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.

[75] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[76] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Design Autom. Electr. Syst.*, 4(3):257–279, 1999.

[77] C. Linn and S. K. Debray. Obfuscation of executable code to improve resistance to static disassembly. *ACM Conference on Computer and Communications Security*, pages 290–299, 2003.

[78] M. Lv, G. Nan, W. Yi, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. *IEEE Real-Time Systems Symposium*, 2010. forthcoming.

[79] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[80] K. L. McMillan. Applications of craig interpolants in model checking. *TACAS*, pages 1–12, 2005.

[81] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of Verilog. *MEMOCODE'10*, pages 179–188. IEEE, 2010.

[82] P. O. Meredith, M. Katelman, J. Meseguer, and G. Rosu. A formal executable semantics of verilog. *MEMOCODE*, pages 179–188, 2010.

[83] J. Meseguer. Rewriting as a unified model of concurrency. *CONCUR*, pages 384–400, 1990.

[84] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. *CONCUR*, pages 331–372, 1996.

[85] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstraction. *Automated Deduction CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003.

[86] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Electronic Notes in Theoretical Computer Science*, 156(1):27–56, 2006.

[87] A. Metzner. Why model checking can improve WCET analysis. *CAV*, pages 334–347, 2004.

[88] M. Montenegro, R. Peña, and C. Segura. A space consumption analysis by abstract interpretation. *FOPARA*, pages 34–50, 2009.

[89] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[90] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language. *In Second Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[91] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. *Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing*, HVC'07, pages 185–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[92] G. Norman, D. Parker, M. Z. Kwiatkowska, S. K. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Asp. Comput.*, 17(2):160–176, 2005.

[93] P. C. Ölveczky. Semantics, simulation, and formal analysis of modeling languages for embedded systems in real-time maude. *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 368–402, 2011.

[94] C. Y. Park. *Predicting deterministic execution times of real-time programs*. PhD thesis, Seattle, WA, USA, 1992.

[95] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

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

[97] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. *Symposium on Programming*, pages 337–351, 1982.

[98] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.*, 4(4):751–778, 2005.

[99] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[100] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. *AMAST '10*. LNCS, 2010.

[101] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[102] G. Rosu and A. Stefanescu. Towards a unified theory of operational and axiomatic semantics. *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, volume 7392 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2012.

[103] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. *IN ACM LCTES*, pages 35–44, 1999.

[104] T. F. Şerbănuţă. *A rewriting approach to concurrent programming language design and semantics*. PhD thesis, 2011.

[105] T.-F. Serbanuta, G. Rosu, and J. Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

[106] N. Shankar. Rewriting, inference, and proof. *Workshop on Rewriting Logic and Applications*, pages 1–14, 2010.

[107] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[108] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.

[109] L. Thiele and R. Wilhelm. Design for time-predictability. *Design of Systems with Predictable Behaviour*, 2004.

[110] M. Webster and G. Malcolm. Detection of metamorphic and virtualization-based malware using algebraic specification. *Journal in Computer Virology*, 5(3):221–245, 2009.

[111] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing analysis for data caches and set-associative caches. *IEEE Real Time Technology and Applications Symposium*, pages 192–202, 1997.

[112] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. *VMCAI*, pages 309–322, 2004.

[113] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[114] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[115] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. *CAV*, pages 22–36, 2008.

[116] J. Yan and W. Zhang. Analyzing the worst-case execution time for instruction caches with prefetching. *ACM Trans. Embedded Comput. Syst.*, 8(1), 2008.

[117] J. Yan and W. Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 80–89. IEEE Computer Society, 2008.

[118] X. Zhang and R. Gupta. Whole execution traces. *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 105–116. IEEE Computer Society, 2004.