Using the Executable Semantics for CFG Extraction and Unfolding

Mihail Asăvoae, Irina Măriuca Asăvoae
Faculty of Computer Science, Alexandru Ioan Cuza University
Iaşi, Romania
{mihail.asavoae, mariuca.asavoae}@info.uaic.ro

***Abstract*—The longest path search problem is particularly important in the context of low-level worst-case execution time analysis. This implies that all program executions are exhibited and inspected, via convenient abstractions, for their timing behavior. In this paper we present a definitional program unfolder, that is based on the formal executable semantics of a target language. We work with $\mathbb{K}$, a rewrite-based framework for the design and analysis of programming languages. Our methodology has two phases. First, it extracts, via reachability analysis, a safe control-flow graph (CFG) approximation, directly from the executable semantics of the language. Second, it unfolds the control-flow graph, annotated with loop bounds, and outputs the set of all possible program executions. The two-phased methodology describes, what we call, a definitional program unfolder and is implemented using the $\mathbb{K}$-Maude tool, a prototype implementation of the $\mathbb{K}$ framework.**

I. Introduction and Related Work

Knowledge of program execution time bounds is important in the context of design and verification of embedded real-time systems. Such systems interact with the external environment, yielding a set of real-time constraints that ensures the correctness of the design. In the verification process, it is important to know a priori tight upper bounds on worst-case execution time (WCET) [1] of hard real-time software components of the system. In general, the problem at hand is the WCET estimation of a given program on a given processor. Thus, two important issues should be addressed: the longest path search and the micro-architecture modeling. With respect to the former, the longest path analysis returns the sequence of instructions that will be executed in the worst case scenario, with one of the most successful approaches being the use of integer linear programming (ILP) as in [2].

We propose a new approach to generate all executable paths of a program via unfolding the semantics specification of the language and using the $\mathbb{K}$ framework [3]. In this way, we create the premises to apply useful abstractions to prune the search space while collecting timing information. We start with the $\mathbb{K}$ definition of a RISC assembly language, a subset of which is briefly introduced in [4], and we (1) explore this definition to extract control-flow information and (2) unfold the result using manual loop bound annotations as well as further structural-related assertions. The result of (1) is a safe over-approximation of the control flow graph which is further unfolded using the state-space exploration capabilities of the Maude system.

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 augmented semantic rules are those of branch and jump instructions. The second phase uses the over-approximation of the CFG computed during the first phase and a set of loop bound annotations. The concrete semantics of the language is employed to define the simplified abstract semantics by using symbolic data values instead of real data values. The definitional program unfolder outputs a trace semantics of 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.

In this work we 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 in order not to 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. A branch instruction is an entry loop point for exactly one loop. The current design and implementation is amenable to further extensions.

Related Work. Li, Malik and Wolfe [5] model the control flow graph as an integer linear program in 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 the cost function. The cost function is the number of CPU cycles the program takes when executed, and it needs to be maximized.

The Maude system [6] 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, as well as other specialized checkers, it enables the specification and analysis of programming languages. The $\mathbb{K}$ framework, described in [3], 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 [7], Verilog in [8], as well as definitions for type systems or a Hoare style program verifier [9]. The $\mathbb{K}$-Maude tool [10] implements the $\mathbb{K}$ framework on top of Maude system and provides, in this way, access to all Maude's aforementioned supporting tools. With respect to rewriting logic in general and $\mathbb{K}$ in particular, this is a continuation of our first use of rewriting-based techniques for WCET analysis in [4]. Whereas [4] proposes an abstraction schema with two simple data abstractions as examples, the current work focuses only on the control flow abstractions exemplified by the CFG extraction and unfolding.

In the context of control-flow graph extraction methodologies, [11] 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). [12] 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. [13] 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 [14], and in verification as described in [15]. Our unfolder generates execution paths having the entire state of the program together with the timing information.

Outline of the paper. The paper is organized as follows: Section 3 presents the $\mathbb{K}$ definitions for the concrete formal executable semantics and how to derive from this an abstract semantics to collect and use control-flow information. Section 4 describes the technology we use to perform reachability analysis on the derivation. The last section contains the conclusions.

II. The Abstract Semantics for CFG Extraction

The $\mathbb{K}$ framework is a rewrite-based framework specialized in the design and analysis of programming languages. A $\mathbb{K}$ specification consists of *configurations*, *computations*, and *rules*. The configurations, formed of $\mathbb{K}$ cells, are labeled and nested structures that represent program states. The rules in $\mathbb{K}$ are divided into two classes: *computational rules* that may be interpreted as transitions in a program execution, and *structural rules* that modify a term to enable the application of a computational rule. The $\mathbb{K}$ framework enables the modular and executable semantics specification of programming languages. We reuse the $\mathbb{K}$ specification of the assembly language by simply adding a new cell called `cfg` to capture information about the (abstract) control flow graph. The elements of `cfg` cell are edges of the form $src \mapsto dst$, where src and dst represent the source and destination program points. The abstract configuration *CFGConfig*, presented next, is based on the language configuration, *LangConfig*.

$$\mathit{CFGConfig} \equiv LangConfig\ \langle \mathsf{Map}[\mathit{Int32} \mapsto \mathit{Int32}] \rangle_{\mathsf{cfg}}$$

We describe next the first phase of our definitional program unfolder. The `cfg` cell is populated with edges upon the program execution, according to the rules in Figure 1 and using the state space exploration tool from Maude. We design this set of rules targeting an increased modularity such that to minimize the number of changes in the semantic definition of the language. In this way, out of all the *RISC* instructions, only the jump and branch instructions (which alter the normal program flow) 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 Figure 1, and we add the corresponding edge in the `cfg` cell during the instruction fetch phase. 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 `syint32`. 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 for all possible execution traces.

III. The Abstract Semantics for Program Unfolding

The unfolding phase uses the CFG, computed in the previous phase, together with manual annotations for loop bounds, represented also in a $\mathbb{K}$ cell called `loops`, and a loop stack cell called `lstack` which keeps track of the current state of the unfolding. The unfolder configuration is:

$$\mathit{UfldConfig} \equiv \mathit{CFGConfig}\quad \langle \mathsf{Map}[\mathit{Int32} \mapsto \mathit{Loop}] \rangle_{\mathsf{loops}}$$
$$\langle \mathsf{List}(\mathit{Int32}, \mathit{Int32}) \rangle_{\mathsf{lstack}}$$

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

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

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

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

$$\text{RULE: } \langle \frac{\texttt{jr}\ Rs;}{\texttt{setPC(}1, Rs)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle \cdots \frac{PC -_{Int32} 4 \mapsto PC}{PC -_{Int32} 4 \mapsto \mathsf{syint32}} \cdots \rangle_{\mathsf{cfg}}$$

Figure 1. CFG Extraction Rules

points of interest: the first and the last executed instructions in the loop, and the first instruction executed upon loop exit. 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 Figure 2. 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.

All the program unfolder rules use an semantic construct defined by the term `geti`(PC), and depend on loop nesting and bound values. The first 3 rules cover the following case: rule [1] handles the simple case of programs without loops, rule [2] handles the first instruction in programs with loops, while rule [3] handles an instruction that is not the first nor the last executed in the loops. When the current executed instruction is also a loop test, we distinguish several situations, as in rules [4-6]. First, in rule [4], the unfolding process reaches a loop and pushes its start instruction and its bound onto the stack. Rule [5] covers the case of getting to an inner loop and pushing into the stack its starting program point and the number of iterations, while rule [6] handles having more iterations for the current loop unfolding. The process of unfolding could exit a particular loop at any iteration, and rule [7] emphasizes this case by popping the loop stack. The last two rules present the jump back caused by the last executed instruction of a loop, and the execution of the instruction in the loop that follows immediately after the loop test. The number of remaining iterations for this particular loop is decremented in the afferent rule [9].

The unfolding rules in Figure 2 leave the concrete semantics unmodified, as they interact with the semantics specification through the term `geti`. One could notice that the `lstack` cell actually plays the role of an abstract `k` cell. This observation leads to an alternative definition of the unfolder, one that moves the loop bounds management from the `lstack` into the actual `k` cell. Therefore, the concrete semantics rules for branch instructions, as the loop entries, needs to maintain information about the corresponding loop bounds. Due to lack of space, we do not cover any further this alternative definition.

IV. Reachability Analysis for Program Unfolding

We work with the current implementation of the $\mathbb{K}$ framework, called $\mathbb{K}$-Maude. This is developed on top of Maude system and it has access to all verification tools that Maude offers. We choose the reachability analysis tool on both steps of our definitional unfolder. $\mathbb{K}$-Maude takes $\mathbb{K}$ specifications and generates rewrite theories in Maude. A rewrite theory has an underlying equational theory containing equations and membership statements, plus rewrite rules. A rewrite theory defines an abstract transitional system with the equations giving, via equivalence classes, the states of the system while the rewrite rules give the transitions in the system. If the lefthand 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 righthand side of the particular rewrite rule. Maude system offers the possibility of unfolding this transition system and proving properties or getting counterexample information.

The transition system provided by a finite rewrite theory is unfolded using the special `search` command from Maude. The class of hard real-time programs for which any program execution terminates is represented by a finite rewrite theory. In our formal semantics, 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. We use `search` command for the CFG extraction, mainly to refine the possible set of target addresses of an indirect jump instruction. The unfolding of the annotated CFG is implemented by the $\mathbb{K}$ rules described in the Figure 2. This phase yields a potential exponential number of program execution paths. We use the unfolder to generate the search space for which the loop annotations hold, and we need further abstractions to smartly traverse and compute timing bounds.

V. Conclusions

Computation of accurate timing bounds for embedded programs requires an assembly language-level analysis of the code. The executable files are disassembled and algorithms for control flow graph extractions are applied.

RULE [1]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle \cdot \rangle_{\mathsf{loops}}$

RULE [2]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle PC \mapsto PC_2\ \mathit{CFG} \rangle_{\mathsf{cfg}}\ \langle L \rangle_{\mathsf{loops}}$ when $\texttt{chkFst}(PC, \mathit{CFG}) \wedge_{\mathit{Bool}} \texttt{noBnd}(PC, L)$

RULE [3]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle PC_2 \mapsto PC\ \mathit{CFG} \rangle_{\mathsf{cfg}}\ \langle L \rangle_{\mathsf{loops}}$ when $\texttt{notIn}(\mathit{CFG}, PC_2) \wedge_{\mathit{Bool}} \texttt{noBnd}(PC, L)$

RULE [4]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle \cdots PC \mapsto \texttt{loop}(\_, \_, B) \cdots \rangle_{\mathsf{loops}}\ \langle \frac{\cdot}{(PC, B)} \rangle_{\mathsf{lstack}}$

RULE [5]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle \cdots PC \mapsto \texttt{loop}(\_, \_, B) \cdots \rangle_{\mathsf{loops}}\ \langle \frac{(PC_1, B_1)}{(PC, B)\ (PC_1, B_1)} \cdots \rangle_{\mathsf{lstack}}$ when $PC_1 \neq_{\mathit{Int32}} PC$

RULE [6]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle \cdots PC \mapsto \texttt{loop}(\_, \_, \_) \cdots \rangle_{\mathsf{loops}}\ \langle (PC, B) \cdots \rangle_{\mathsf{lstack}}$ when $B \geq_{\mathit{Int32}} 0$

RULE [7]: $\langle \frac{\cdot}{\texttt{geti}(PC_1)} \rangle_{\mathsf{k}}\ \langle PC_1 \rangle_{\mathsf{pc}}\ \langle \cdots PC \mapsto PC_1\ PC \mapsto PC_2 \cdots \rangle_{\mathsf{cfg}}\ \langle \cdots PC \mapsto \texttt{loop}(\_, PC_2, \_) \cdots \rangle_{\mathsf{loops}}\ \langle \frac{(PC, B)}{\cdot} \cdots \rangle_{\mathsf{lstack}}$
when $B >_{\mathit{Int32}} 0$

RULE [8]: $\langle \frac{\cdot}{\texttt{geti}(PC_2)} \rangle_{\mathsf{k}}\ \langle PC_2 \rangle_{\mathsf{pc}}\ \langle \cdots PC \mapsto \_\ PC \mapsto PC_2 \cdots \rangle_{\mathsf{cfg}}\ \langle L\ PC \mapsto \texttt{loop}(\_, PC_2, \_) \rangle_{\mathsf{loops}}\ \langle (PC, B) \cdots \rangle_{\mathsf{lstack}}$
when $B >_{\mathit{Int32}} 0 \wedge_{\mathit{Bool}} PC_2\ \texttt{notIn}\ L$

RULE [9]: $\langle \frac{\cdot}{\texttt{geti}(PC)} \rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle \cdots PC_1 \mapsto \texttt{loop}(PC, \_, \_) \cdots \rangle_{\mathsf{loops}}\ \langle \frac{(PC_1, B)}{(PC_1, B -_{\mathit{Int32}} 1)} \cdots \rangle_{\mathsf{lstack}}$ when $B >_{\mathit{Int32}} 0$

Figure 2. CFG-based Unfolding Rules

In the presence of jump instructions with dynamically computed addresses, such algorithms compute safe over-approximations of the actual CFG.

In the context of a new methodology for WCET analysis of hard-real time programs, centered around an assembly language definition, we present 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 define a semantic program unfolder that is used to compute the set of all possible program executions. The method assumes a number of restrictions in terms of annotations, and GFG structure. 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 from a low-level programming language.

As future work, we propose to define a data analysis to compute bounds for program variables and to use these for improving the approximation of the CFG. With respect to the program unfolder, we propose to improve the assertion language and to introduce further control-flow related constraints. These, corroborated with work on micro-architecture modeling should conduct to the first WCET analyzer using the rewrite-based technology.

Acknowledgment

We thank Professor Dorel Lucanu as well as to anonymous reviewers for the valuable comments and advices that helped us to improve this work. This work was supported by 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.

References

[1] 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.*, vol. 7, no. 3, pp. 1–53, 2008.

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

[3] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[4] M. Asavoae, D. Lucanu, and G. Roşu, "Towards semantics-based WCET analysis," *WCET*, 2011.

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

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

[7] C. Ellison and G. Roşu, "A formal semantics of C with applications," University of Illinois, Tech. Rep. http://hdl.handle.net/2142/17414, November 2010.

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

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

[10] T. F. Şerbanuţă and G. Roşu, "K-Maude: A rewriting based tool for semantics of programming languages," in *WRLA 2010*, ser. LNCS, vol. 6381, 2010, pp. 104–122.

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

[12] D. Kästner and S. Wilhelm, "Generic control flow reconstruction from assembly code," in *LCTES*. ACM, 2002, pp. 46–55.

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

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

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