Context-Updates Analysis and Refinement in Chisel*

Irina Măriuca Asăvoae[1], Mihail Asăvoae[2], Adrián Riesco[3]

[1] Swansea University, UK
[2] CEA LIST, France
[3] Universidad Complutense de Madrid, Spain

Abstract. This paper presents the context-updates synthesis component of Chisel, a tool that synthesizes a program slicer directly from a given algebraic specification of a programming language operational semantics. By context-updates we understand programming language constructs that induce unconditional control-flow non-sequentiality, i.e., gotos or subroutine calls. The context-updates synthesis follows two directions: an overapproximating phase that extracts a set of potential context-update constructs and an underapproximating phase that refines the results of the first step by testing the behavior of the context-updates constructs produced at the previous phase. We use two experimental semantics that cover two types of language paradigms: high-level imperative and low-level assembly languages and we conduct the tests on standard benchmarks used in avionics.

Keywords: generic slicing tool, programming languages formal semantics, Maude, synthesis

1 Introduction

Slicing is a program analysis technique that takes a program and a *slicing criterion* (i.e., a set of variables V) and produces a *program slice* (i.e., the program parts containing language construct units that may directly or indirectly change during execution the variables in V). We refer to the language construct units, i.e., the syntactic components of the programming language, separated by sequencing operators, as *instructions*. In this paper, we focus on static slicing, i.e., when the slices are computed without executing the program, and we refer to it as simply *slicing*.

Program slicing relies on the evaluation of the data-flow equations over the control-flow graph of the program. Obviously, besides the data-flow, there is a need for additional techniques to deal with other language features. In [26] we find a comprehensive survey on the standard program slicing techniques applied over different programming language concepts such as standard imperative,

* This research has been partially supported by the MINECO Spanish project *TRACES* (TIN2015-67522-C3-3-R) and by the Comunidad de Madrid project *N-Greens Software-CM* (S2013/ICE-2731)

pointers, unstructured control flow, and concurrency. Generally, these techniques use an augmented control-flow graph, e.g., the function calls are usually represented by edges [24] in a call graph.

Meanwhile, *the rewriting logic semantics project* [14] promotes the programming languages semantics are defined as rewriting systems using Maude [6], and it is followed by the work in the $\mathbb{K}$ framework [23]. Our work complements the rewriting logic semantics project by developing static analysis methods, in particular slicing, for programs written in languages with an already defined rewriting logic semantics in Maude. Our approach analyses a given language semantics and synthesizes the necessary information for program slicing. We use the results of the syntheses to traverse the program term in order to obtain the program slice.

Our approach is implemented in Chisel,[4] a Maude tool for generic program slicing [20]. Chisel takes a programming language semantics, given as a Maude specification, breaks it into pieces of interest for slicing, and uses these pieces to augment the program and to produce the program model, which is then sliced. Chisel synthesizes these semantics to extract operators that produce certain update patterns in the underlying machine model. These operators are then used to produce necessary information for slicing, e.g., side-effect instructions. The final step of Chisel is the program slicing analysis that takes a program and produces its slice w.r.t. a slicing criterion. With Chisel we target sequential imperative code without dynamic allocation that is generated from synchronous designs—a class of applications used in real-time systems, e.g., avionics. We experiment for now two semantics: one for an imperative language with functions, WhileFun [9,3], and one for the MIPS assembly language.

Chisel aims to evolve into a framework for *generic static slicing*. The progression in Chisel design and implementation is described in [18,3,19]. In [18] we present the methodology for performing intraprocedural slicing, which is improved in [3] by implementing interprocedural slicing. In [19] we introduce an algorithm for inferring the data-flow information to automatically detect how the language constructs work with the memory.

The contribution of this paper is presenting the context-updates synthesis component of Chisel, where by context-updates we understand programming language constructs that unconditionally produce non-sequential change in the control-flow, such as goto instructions or function calls. The main motivation for our work is the fact that for the interprocedural slicing algorithm implemented in Chisel we need to identify the function calls in order to produce the appropriate control-flow. Until now we gave this information manually, as user input to the slicing component of Chisel. With the current work we improve the genericity of Chisel by automatic synthesis of context-updates.

The context-updates synthesis follows two directions: an overapproximating phase when we analyze the language semantics specification to extract a set of potential context-update constructs and an underapproximating phase when we stress-test the semantics to refine the context-updates obtained at the first

[4] `https://github.com/ariesco/chisel`

step. The underapproximating phase, introduced in this paper, is justified by the imprecision of the overapproximating phase for the context-updates. The imprecision is mostly due to the laxity of the automatic detection of stack-like memory operators.

The rest of the paper is organized as follows: Section 2 presents the related works; Section 3 overviews the Chisel system; Section 4 defines and characterizes the context-updates; Sections 5 and 6 describe the context-updates synthesis and respectively context-updates refinement and evaluation. Section 7 concludes and outlines future work directions.

2 Related Work

Program slicing [27] is a standard analysis technique used to compute program slices based on certain criteria for program input. Slicing without program execution input is called static slicing and slicing based on execution of specific program inputs is called dynamic slicing.

Generic slicing. Techniques of generic program slicing are proposed in [5,7]. The program slicing of [7] uses an algorithm that extracts slices from a common intermediate representation named PIM, however it requires a non-trivial, language dependent transformation between a particular language and PIM. The work in [7] is generic because it uses a notion of constraints slices to represent both static and dynamic slices and transforms various slicing methods into instances of a parametric slicing procedure. Chisel, extended with the proposed method of context-updates synthesis considers only static slicing and addresses genericity from a different angle: it eliminates the need of a language-dependent translation by working directly on the formal language semantics. Generic program slicing is also the focus in [5]. The ORBS tool [5] proposes a technique for dynamic slicing based on statement deletion. A program slice is iteratively constructed by removing statements from the original program and then checking if the transformation is semantics-preserving w.r.t. the slicing criterion. Checking the semantics preservation relies on novel testing techniques [13]. The static slicing of Chisel complements the dynamic slicing of ORBS, as it computes static program slices based on in-depth investigation of the formal language semantics. And as in [13], Chisel, through the current work, integrates testing based on path-coverage to improve the precision of the context-updates synthesis, while it remains generic w.r.t. the language semantics.

Environments and context-updates. Functional programming proposes richer notions of contexts and context manipulation than what we consider in our framework. Briefly, the standard definition of a context as variables in scope is extended in functional languages in several directions. On one hand, there are high-level constructs such as *call/cc* - call with current continuation - in the Scheme language [1], where snapshots of the current control states are manipulated as values (e.g., passed as arguments to function calls). On the other hand, there are extended notions of contexts to capture security properties, as in the SLam calculus [8] or parameters of execution platforms [25,16]. Such contexts

are used to track how programs affect an execution environment (e.g., the effect systems [25]) or how programs depend on the execution environment (e.g., the coeffect systems [16]). In our work, the context is a first-order variable that could be explicitly or implicitly represented in a programming language semantics. We identify context changes (i.e., context-updates) in a generic manner, directly from a formal language semantics given as rewrite theories. Our context-updates synthesis is more general as the aforementioned work in functional programming, due to the genericity of our approach, i.e., we do not address a particular type of memory/environment representation as the one in functional programming. Nevertheless, the rich context representations from functional programming could be used to specialize our context-updates synthesis with the inference of types of variables updates during context changes.

Formal semantics and testing. The rewriting logic semantics project [14] advocates for specialized modeling and reasoning techniques based on formal semantics of programming languages. In this direction, the $\mathbb{K}$ framework [21] relies on a convenient notation to advance the development of formal language definitions and it argues that such language definitions should be used directly, as they are, in tool construction for program reasoning. Apart from its intrinsic specification strengths, the $\mathbb{K}$ framework proposes a verification environment called matching logic [22]. Matching logic uses patterns to specify properties over the program state space and employs symbolic execution and pattern matching to validate program properties. In this context, program verification with matching logic means to symbolically run the target program on the formal semantics, while the generic techniques behind Chisel allow the symbolic analysis the formal semantics in order to synthesize the necessary semantic ingredients to address the program properties. The approach in [2] implements dynamic slicing for execution traces of the Maude model checker. The semantics is executed for an initial given state, then dependency relations are computed using a backward tracing mechanism. In comparison, our approach focuses on statically computing slices for programs and not for given traces (e.g., of model checker runs). The tool in [17] implements a technique of rule-coverage testing as it generates test cases from the formal semantics of programming languages given as rewrite theories. Specifically, the semantic rules are employed to instantiate the variables used by the given program based on a narrowing technique. The refinement phase of our context-updates synthesis uses path-coverage testing provided by PathCrawler [12], a specialized plugin of the Frama-C analyzer [11]. PathCrawler generates test sets for ANSI C code using a combination of symbolic execution and constraint solving capabilities to ensure complete path-coverage. The current work uses test-coverage with PathCrawler to refine the context-updates synthesis of Chisel as opposed to the aforementioned usage of rule-coverage, in [4] (which tests individual language constructs but it is agnostic w.r.t. the program semantics).

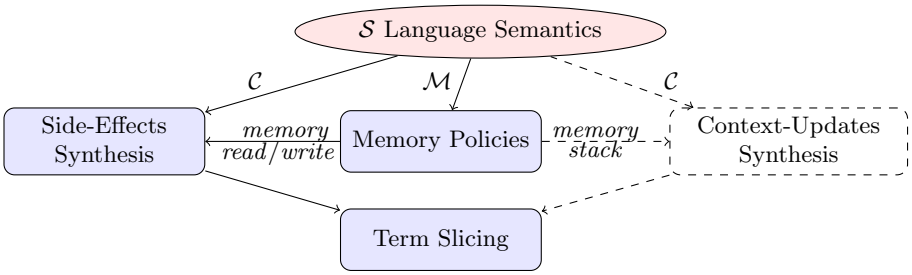


Fig. 1. Chisel components: the formal language semantics and the syntheses.

3 The Chisel System

We briefly describe in this section the ideas underlying Chisel that aims to synthesize slicers from programming language semantics specifications. A summary of Chisel is presented in [20] where Fig. 1 summarizes the design structure of the tool. Namely, we denote by $\mathcal{S}$ the Chisel input, i.e., the semantics specification of a programming language defined in Maude. The memory policies work over $\mathcal{M}$—the memory component of $\mathcal{S}$, while the syntheses examine $\mathcal{C}$—the language syntax component of $\mathcal{S}$. Next we describe in Fig. 1 by levels, i.e, the structure of $\mathcal{S}$, the syntheses that work over components of $\mathcal{S}$, and the term slicing that synthesizes the slicer based on the results obtained from the above level.

3.1 Programming language semantics

The programming language semantics specification $\mathcal{S}$ is a Maude theory with the structure $\mathcal{C} \implies \overrightarrow{\mathcal{M}}$ where:

- $\mathcal{C}$ represents the context free grammar defining the language syntax;
- $\mathcal{M}$ defines the machine on which the programs are executed; $\mathcal{M}$ is formed by components such as static and dynamic memory where the program variables are assigned values directly or indirectly, program code section, and so on;
- $\implies$ is the set of rules $< C, M > \Rightarrow < C', M' >$ defining how the machine state $M \in \mathcal{M}$ is changed into $M' \in \mathcal{M}$ by the syntactic construct(s) $C \in \mathcal{C}$, while $C' \in \mathcal{C}$ represents the syntactic construct(s) that continue the computation;
- $\longrightarrow$ over the $\mathcal{M}$ component denote the machine operations, e.g., setting the value of a variable.

Note that the context free grammar production rules as $S \to S_1 t_1 \dots S_n t_n$ are defined in Maude by operators `op` $t_1\_\dots t_n\_ : S_1 \dots S_n$`->`$S$ where $t_i, 1 \leq i \leq n \in \mathbb{N}$ are terminal symbols and $S, S_i, \leq i \leq n$ are nonterminals defined by sorts/types in Maude. The sorts sequence $S_1 \dots S_n$ is called the arity of the operator while S is the co-arity.

In $\mathcal{S}$ we assume that the syntax has a top nonterminal denoted by $\mathcal{C}$ that is the sort representing the co-arity of the language constructs. For example a branching instruction is defined as `op If_Then_Else_ :` $B\ \mathcal{C}\ \mathcal{C}$ `->` $\mathcal{C}$, where B is the top sort for Boolean expressions. (Note that the arity of this operator is

$B\ \mathcal{C}\ \mathcal{C}$, while the co-arity is $\mathcal{C}$.) Hence, the language instructions are assumed to be defined in $\mathcal{S}$ by operators with the $\mathcal{C}$ co-arity. Consequently, the context free grammar denoted by $\mathcal{C}$ is equipped with a parser provided by Maude that transforms a program p into the associated parse tree t_p which in rewriting is called a *term* (of sort $\mathcal{C}$). Also, an example of rules in $\Longrightarrow$ denoting the command of the branching instruction is:

```
crl [If1] : < If be Then C Else C', m > => < skip, m'' >
 if < be, m > => < T, m' > /\ < C, m' > => < skip, m'' > .
crl [If2] : < If be Then C Else C', m > => < skip, m'' >
 if < be, m > => < F, m' > /\ < C', m' > => < skip, m'' > .
```

where the rule labeled `[If1]` is executing the `Then` branch containing the code C if the condition `be` evaluates in the memory `m` to the true value `T` and the rule labeled `[If2]` is executing the `Else` branch containing the code C' if the condition `be` evaluates in the memory `m` to the false value `F`.

The machine representation in $\mathcal{S}$ is given by $\overrightarrow{\mathcal{M}}$ where, for example, the static memory component is defined by an operator which lists elements of sort W representing pairs of the variables (of sort *Var*) with the values (of sort *Val*). Note that the pairs are defined by another operator of arity *Var Val* and co-arity W. The changes in the machine are denoted by $\longrightarrow$ representing a set of operators and their associated equations or rewrite rules that specify how the machine internals are working. For example, a variable look-up in the static memory is defined by an operator `op _[_] :` *W Var* `->` *Val* defined by an equation or a rewrite rule that identifies the pair pr in the static memory $w : W$ containing the variable v : *Var* as first element and returns the second element of pr that represents the value n : *Val* associated to v in w.

3.2 Memory policies and syntheses

By a memory policy mp we understand a subset of memory operators $\mathcal{M}(mp) = \{o : w \to s \mid o \in \mathcal{M} \text{ s.t. } mp(\overrightarrow{o})\}$, where mp is a property of the arity and co-arity of o and of $\overrightarrow{o}$, i.e., the rules matching the operator o. Note that s denotes a sort while w denotes a list of sorts. Also, given a rewrite rule or equation $[r]$: $lhs \to rhs$ where r is the rule label, lhs, rhs are terms in $\mathcal{M}$ then $r \in \overrightarrow{o}$ iff the operator o is a subterm of lhs. For example, *memory-read* policy $\mathcal{M}(read)$ is the set of operators in $\mathcal{M}$ that contain in their arity the sort for variables *Var* and for static memory W and in their co-arity the sort for values *Val*. A *memory-write* operator $o_w \in \mathcal{M}(write)$ contains in its arity sorts for static memory W, variables *Var*, and values *Val*, and in its co-arity the static memory sort W. Moreover, any $r \in \overrightarrow{o_w}$ replaces in rhs the value parameter of the operator o_w from lhs into the static memory pair containing the variable. The read/write memory policies are presented in [19].

The syntheses analyze $\Longrightarrow$, the semantics rules in $\mathcal{S}$, in order to identify the subset of instructions in $\mathcal{C}$ that induce a certain change pattern in the memory. The memory change pattern is defined by a combination of memory policies.

The main assumption of the synthesis methodology is that rules $r_{\mathcal{C}}$ in $\Longrightarrow$ trigger rules $r_{\mathcal{M}}$ in $\overrightarrow{\mathcal{M}}$. If $r_{\mathcal{M}} \in \overrightarrow{\mathcal{M}}$ follows the assumed memory change pattern (i.e., the memory policies combination) then the instruction $i \in \mathcal{C}$ that matches the triggering rule $r_{\mathcal{C}} \in\Longrightarrow$ is selected by the synthesis. For example, in [18] we describe the side-effects synthesis defined as the set of instructions i that *may* trigger a memory write for some variable subterm in i, which we denote as destination variable, and memory reads for other subterms, which we denote as sources. In [19] we describe a follow-up methodology that identifies the source-destination subterms in i by following backwards the subterm dependencies between $r_{\mathcal{M}}$ and $r_{\mathcal{C}}$.

3.3 Term analysis

Program slicing takes as input a program p and a *slicing criterion* S consisting of a set of program variables that are considered the initial side-effect destinations. In the *first step* all the instructions in p that have some element of S as a side-effect destination are added to the slice sp. Next, p is traversed with a fix-point algorithm that adds to S the side-effect sources in sp and repeats the first step until S is saturated. In Chisel we implement a state-of-the-art interprocedural slicing algorithm introduced in [10] that uses call graph information to produce a more precise slice sp. This part of Chisel is described in [3]. However, for preserving the genericity of the tool, the call graph had to be inferred from $\mathcal{S}$. Until now we assumed the call graph node patterns as given by the user where by *call graph node patterns* we understand the function call instructions that represent the source nodes in the call graph. In the current work we give the methodology for identifying call graph node patterns as context-updates.

4 Context-updates Definition and Characterizations

In this section we give the definition of context-updates and present two (static and dynamic) characterizations for this notion. The static characterization analyzes $\mathcal{S}$ to extract the context-updates while the dynamic characterization executes the semantics $\mathcal{S}$ for the same purpose.

Definition 1. *A context-update is a ground term of $\mathcal{S}$ that instantiates an instruction which identifies a unconditional non-sequential control-flow in some program p in $\mathcal{S}$. The context-updates is the set of syntactic constructs that match a context-update in some program:*

$$CU := \{i \in \mathcal{C} \mid \exists p \in \mathcal{C} \text{ a ground term s.t. } \exists \theta \text{ a ground substitution s.t. } \theta(i) \prec p \text{ and } \theta(i) \text{ context-update}\}$$

where a ground substitution replaces the nonterminals in the production rule of the context free grammar denoted by $\mathcal{C}$ with words in the language of $\mathcal{C}$, while given two words w_1, w_2 we have $w_1 \prec w_2$ if $w_2 = w\ w_1\ w'$.

Note that $\mathcal{C}$ is a part of $\mathcal{S}$ hence we use the complementary terminology terms instead of words while $\prec$ is the subterm relation.

The *static characterization* of CU relies on the observation that in imperative languages with local memory (i.e., the class of languages we consider) the function call instructions need to save the local context, i.e., a part of the static memory state. Standardly, the local context is saved on a stack structure due to the fact that the rules $\Longrightarrow$ behave as the transition rules in a pushdown system $\mathcal{P}$ where $\mathcal{M}$ represent the states in $\mathcal{P}$. Our assumptions regarding to the structure of the semantics $\mathcal{S}$, i.e., the separation between the language syntax and the machine, induce the static characterization:

SC: The context-updates of the programming language specified by $\mathcal{S}$ is a subset of instructions in $\mathcal{C}$ that *must* trigger a memory stack change:

$$SC := \{i \in \mathcal{C} \mid \forall r_{\mathcal{C}} \in \Longrightarrow \\ \text{if } i \ll lhs(r_{\mathcal{C}}) \text{ then } \exists r_{\mathcal{M}} \in \mathcal{M}(stack) \text{ s.t. } r_{\mathcal{C}} \text{ triggers } r_{\mathcal{M}}\}$$

where $t_1 \ll t_2$ defines the unification relation, i.e., a set of substitutions ϕ s.t. $\phi(t1)$ is a subterm of t_2 and $\mathcal{M}(stack)$ is the instantiation of $\mathcal{M}(mp)$ with a stack memory policy.

Note that **SC** produces an overapproximation of CU due to the abstract nature of the unification and the stack memory policy.

The *dynamic characterization* of CU relies on testing the semantics $\mathcal{S}$ by executing programs from a test set TP using certain testing technique(s) $TestTech$ to generate input states $In(p)$ for any $p \in TP$. Namely, given a set TP of test programs $p \in \mathcal{C}$, $TestTech : TP\mathcal{S} \rightarrow In(\mathcal{M})$ is a function that for each program $p \in TP$ provides a set of initial states that execute p in $\mathcal{S}$, i.e., $In(p) = \{m_0 \in \mathcal{M} \mid \exists m \in \mathcal{M} :< p, m_0 > \overset{*}{\Rightarrow} < \emptyset, m >\}$. We denote by $\Pi_p = \{\pi_p \mid \exists m_0 \in In(p) \text{ s.t. } \pi_p =< p, m_0 > \overset{*}{\Rightarrow} < \emptyset, m >\}$ the executions traces of the program p from the initial states $In(p)$. Furthermore, we denote $L_p = [i_1 \dots i_n]$ the list of instructions $i_k \in \mathcal{C}, \forall 1 \leq k \leq n$ obtained by a preorder traversal of t_p–the tree term associated by parsing to the program p. The dynamic characterization exploits the fact that $\mathcal{S}$ is an *executable* formal semantics as follows:

DC: Assuming that for any execution trace $\pi \in \Pi_p$ (of length $|\pi|$) exists the segmentation $\pi \overset{L_p}{=} w_1 \dots w_n$ such that w_i is a segment of L_p then the context-updates are the separator instructions:

$$DC := \{i \in \mathcal{C} \mid \exists p \in TP, \forall \pi \in \Pi_p \text{ s.t. } \pi \overset{L_p}{=} w_1 \dots w_n \text{ and } i \in \pi \text{ then} \\ \exists 1 \leq j \leq n, \exists 0 \leq a_j < b_j < |\pi| \text{ s.t.} \\ w_j = \pi(a_j)\pi(a_j + 1) \dots \pi(b_j) \text{ and } i = \pi(b_j)\}$$

Hence, DC is the set of instructions that appear at the end of some segment w of L_p in every program execution π that contains them.

Note that **DC** produces an underapproximation of CU due to the fact that TP is a subset of all programs in $\mathcal{C}$ and $TestTech$ produces a subset of all possible program executions. In Sections 5 and 6 we present details of **SC** and **DC**, respectively.

5 Context-updates Synthesis

In this section we present our approach towards discovering context-updates based on their static characterization **SC**.

The methodology we propose for context-updates overapproximation follows the methodology described in the Section 3.2. Namely, we firstly apply sort-based patterns to define a memory policy that identifies stack structures/memory operators or, short, *memory-stacks*. Secondly, using the memory-stacks we construct a tree $\mathcal{T}$ based on $\Longrightarrow$ to discover the set $\mathcal{O}$ of language constructs that must use the memory-stacks. We call $\mathcal{T}$ a hyper-tree and we give in this section an example of such hyper-tree.

The stack memory policy determines $\mathcal{M}(stack)$ where $stack(\overrightarrow{o})$ property first requests that o is a non-commutative operator with the arity $S\ S$ and co-arity S, where S is a sort in $\mathcal{M}$. Moreover, we have two patterns we search for: explicit and implicit. The explicit memory-stack policy requests that all the rules in $\overrightarrow{o}$) either add or substract one element. The implicit pattern uses the conditional rules over the language semantics to produce memory-stacks. The implicit pattern is produced by the Maude's evaluation semantics that uses a an evaluation stack for conditional rules. Namely, the evaluation of the conditional rule's body (i.e., the statement between the `crl` and `if` keywords) is postponed until the evaluation of the rule's condition (i.e., the statement after `if` keyword) is completed.

Example 1. We present in this example the memory specification for WhileFun– an imperative language with assignment, conditional, loops, local variables, an input/output buffer, and function calls [9,3]. Assuming we have defined the syntax for the language in a module `WHILE-SYNTAX` (which includes definitions for variables, Boolean values, and numeric values), the module `MEMORY` imports this module and defines the sorts `Env` for the environment, which maps variables to values, and `ESt` for a stack of environments, which will be used when a new context is required:

```
fmod MEMORY is
  pr WHILE-SYNTAX .
  sorts Env ESt .
  subsort Env < ESt .
  ...
```

where the `subsort` indicates that a single environment states for a singleton stack, i.e., the environment type is a subtype of the environments' stack . Constructors of these sorts are defined by using `op` and the attribute `ctor`. In this case, we define the empty environment (`mt`); a single assignment, which receives a variable and a value (underscores are placeholders); and the composition of environment, defined with empty syntax and defined as commutative and associative and having `mt` as identity:

```
  op mt : -> Env [ctor] .
  op _=_ : Variable Value -> Env [ctor] .
  op __ : Env Env -> Env [ctor comm assoc id: mt] .
```

Similarly, the stack is built by putting together stacks with the `_|_` operator:

```
op _|_ : ESt ESt -> ESt [ctor assoc] .
```

The operator `_|_` follows the explicit memory-stack policy and it will be used in the context-update synthesis, as described next in Example 2. The memory module also contains functions for variables' update, variables' look-up, and new variables allocation.

The synthesis of a set $\mathcal{O}$ of context-updates relies on the construction of a hyper-tree of rules and is similar to the side-effects synthesis described in [18]. The difference here is the fact that at the leaves level we now use a different memory policy (the memory-stack policy) to filter the paths leading to context-updates and we use a *must* strategy, i.e., *all* rules in $\Longrightarrow$ matching an instruction have to contain in their associated subtree a memory-stack. The algorithm implementing this in Chisel is defined by the operator `traverseHypertree` in Fig. 2:

```
op traverseHypertree : Module QidSet TermList ContextUpdates HypertreeTraversalResult
                       -> HypertreeTraversalResult .
eq traverseHypertree(M, none, TL, CU, HTR) = HTR .
ceq traverseHypertree(M, Q ; QS, TL, CU, HTR) = traverseHypertree(M, QS, TL, CU, HTR')
 if Q in CU /\
    HTR' := add2orange(Q, HTR) .
ceq traverseHypertree(M, Q ; QS, TL, CU, HTR) = traverseHypertree(M, QS, TL, CU, HTR)
 if traversed?(Q, HTR) .
ceq traverseHypertree(M, Q ; QS, TL, CU, HTR) =
          if allOrange?(HTR') and not emptyHypernode(M, COND, (T, TL))
          then add2orange(Q, HTR')
          else add2olive(Q, HTR')
          fi
 if COND := getCondition(M,Q) /\ not Q in CU /\ not traversed?(Q,HTR) /\ T := getLHS(M,Q) /\
    HTR' := traverseCond(M, COND, (T, TL), CU, setAllOrangeVar(true, HTR)) .

op traverseCond : Module Condition TermList ContextUpdates HypertreeTraversalResult
                  -> HypertreeTraversalResult .
eq traverseCond(M, nil, TL, CU, HTR) = setAllOrangeVar(false, HTR) .
eq traverseCond(M, T = T' /\ COND, TL, CU, HTR) = traverseCond(M, COND, TL, CU, HTR) .
eq traverseCond(M, T := T' /\ COND, TL, CU, HTR) = traverseCond(M, COND, TL, CU, HTR) .
eq traverseCond(M, T : S /\ COND, TL, CU, HTR) = traverseCond(M, COND, TL, CU, HTR) .
ceq traverseCond(M, T => T' /\ COND, TL, CU, HTR) = combineHypernodes(HTR', HTR'')
 if TV := freshTerm(T) /\
    QS := getRulesUnifying(M, TV, getRls(M), TL) /\
    HTR' := traverseHypertree(M, QS, TL, CU, HTR) /\
    HTR'' := traverseCond(M, COND, TL, CU, setAllOrangeVar(true, HTR')) .
```

Fig. 2. The traverseHypertree operator in Chisel.

The operator in Fig. 2 computes the set of basic syntactic language constructs that *must* be context-updates, by inspecting the conditions and the right-hand side of each rewrite rule in $\mathcal{C}$ represented here as `Q`, i.e., the rule label. The operator unfolds the rewrite rules into the hyper-tree $\mathcal{T}$ with children nodes representing lists of rules that unify with subterms of `Q`'s conditions (each subterm of

Q unifies with a particular node). The traversalHypertree operator goes horizontally in $\mathcal{T}$ if there is no subtree rooted in the current Q node. Otherwise, when Q is the root of a subtree in $\mathcal{T}$ (e.g., when the rule Q is conditional), the traversal goes vertically via the operator traverseCond. The traversalHypertree operator assigns each rule label Q to a particular set, either orange or olive, where these sets are defined as follows:

$$\texttt{orangeSet} := \{\texttt{Q} \in \mathit{nodes}(\mathcal{T}) \mid \exists \texttt{Q}' \in \mathit{subtree}(\texttt{Q}, \mathcal{T}) : \texttt{Q}' \in \texttt{ContextUpdates}\}$$
$$\texttt{oliveSet} := \{\texttt{Q} \in \mathit{nodes}(\mathcal{T}) \mid \forall \texttt{Q}' \in \mathit{subtree}(\texttt{Q}, \mathcal{T}) : \texttt{Q}' \notin \texttt{ContextUpdates}\}$$

The orangeSet contains Qs that are the root of a subtree containing context-updates while oliveSet is context-updates free. Note that the termination of the algorithm in Fig. 2 is ensured by the fact that the specification $\mathcal{S}$ has a finite number of rules, and that any rule in $\mathcal{T}$ that was already added to either orange or olive set is not unfolded anymore. We give next an example that provides the intuition about the synthesis process.

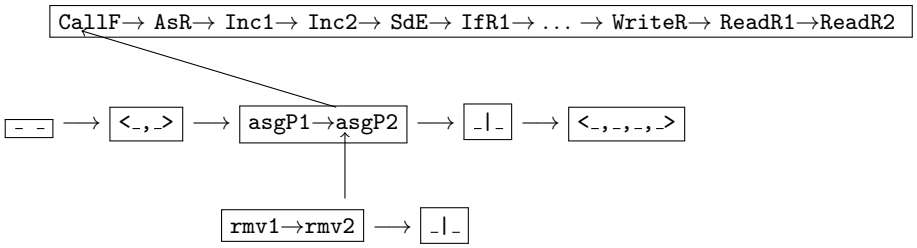


Fig. 3. The hyper-tree constructed for WhileFun.

Example 2. The first part of the hyper-tree $\mathcal{T}_{\text{WhileFun}}$, constructed for WhileFun semantics, is depicted in Fig. 3. The memory-stack operator discovered here at the leaves level is _|_ that is obtained by the explicit memory-stack policy. The root of $\mathcal{T}_{\text{WhileFun}}$ contains the language constructs $\mathcal{C}$ where we show first CallF the rule label that specifies the semantics of a function call such as:

```
crl [CallF] :
 < Call fn(actPrms), st, rwb, fs > => < skip, st'', rwb', fs >
 if fn(Prms){ C } fs' := fs /\ < actPrms, st > => vals /\
    st' := assignPrms(actPrms, Prms, st | mt) /\
    < C, st', rwb, fs > => < skip, st'' | lenv', rwb', fs >  .
```

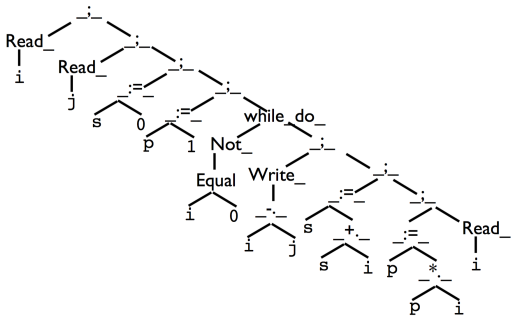
The first condition in the rule CallF extracts the function definition from the function set fs by means of a matching condition; the second condition evaluates the arguments passed to the function; the third condition uses the function assignPrms (listed below) to bind the parameters to the values previously obtained; and the fourth condition evaluates the body of the function in the new stack of environments.

```
Read i ; Read j ;
s := 0 ; p := 1 ;
While Not Equal(i, 0) Do
  Write (i -. j) ; s := s +. i ;
  p := p *. i ; Read i
```

(i)

```
[Read i, Read j, s := 0, p := 1,
 While Not Equal(i, 0) Do _,
 Write (i -. j), s := s +. i,
 p := p *. i, Read i]
```

(iii)



(ii)

Fig. 4. The code list L_p (iii) for the program p (i) and parsed program term t_p (ii).

```
op assignPrms : ExpL VarL ESt -> ESt .
eq [asgP1]  : assignPrms(nv, nv, ro) = ro .
eq [asgP2]  : assignPrms((N,EL), (X,VVs), mu | ro) =
              assignPrms(EL, VVs, mu | remove(ro, X) (X = N)) .
```

6 Context-updates Refinement

This section describes how the dynamic characterisation of context-updates is implemented and used for the refinement of the set $\mathcal{O}$ obtained by the context-updates synthesis.

First, similarly to the stack memory policy, we detect syntactic constructs that produce code sequencing. Namely, a sequencing syntactic construct is defined by a non-commutative operator with arity $\mathcal{C}\ \mathcal{C}$ and co-arity $\mathcal{C}$. For example, in WhileFun a sequencing operator is $\_;\_$.

We recall that the a program p is parsed into a term tree t_p based on context free grammar defined by $\mathcal{C}$. We define the *code list* L_p as the flattening of p into a list of instructions (i.e., unit elements in $\mathcal{C}$) obtained by the preorder traversal of t_p's subtrees that represent the children of some sequencing operator in t_p. In Fig. 4 (i) and (ii) we give an example of a program p and the associated tree t_p, respectively, while (iii) describes the list L_p.

Also, we denote by $[L_p]_{fn}$ the set function definitions in p:

$$\{L_p(k)..L_p(k+n-1) \mid L_p(k) \in \mathcal{C}_{fn} \text{ and } (L_p(k+n) \in \mathcal{C}_{fn} \text{ or } L_p(k+n) = \epsilon) \\ \text{and } \forall i = k+1..k+n-1 : L_p(i) \notin \mathcal{C}_{fn}\}$$

where $L_p(i)$ represents the i-th element of the list L_p and $\mathcal{C}_{fn}$ is the set of program constructs representing function declarations.

Second, given a set of execution traces Π_p of a program p, as defined in Section 4, we denote its elements by ϖ, i.e., an execution path of p w.r.t. $\mathcal{S}$. Furthermore, we denote by π the filtering of ϖ w.r.t. the application of the rules in $\Longrightarrow$. Namely, we only preserve in π the terms ϖ_i which match the lhs of a rule in $\Longrightarrow$. The executions traces ϖ and their filtering into π are obtained using

the Maude debugger tool [17]. We use the standard notation for π, namely $|\pi|$ represents the length of the path, while $\pi_i, i \in \{0, \ldots, |\pi|\}$, represents the i-th element of the path. Note that π_0 is ϵ, the empty execution list.

Definition 2. *The property φ w.r.t. Π_p is defined as follows:*

$$\begin{array}{l}\forall \varsigma \in \mathcal{O},\ \forall \varpi \in \Pi_p, \pi := \mathit{filter}_{\mathcal{C}}(\varpi),\ \forall i \in 1..|\pi| : \pi_i = \varsigma \implies \\ \quad (\pi_{i-1}\pi_i \in L_p \implies \varsigma \in \mathcal{O}_r) \wedge \\ \quad (\pi_{i-1}\pi_i \notin L_p \wedge (\pi_{i-1}, \pi_i) \in [L_p]_{fn}) \implies \varsigma \in \mathcal{O}_g) \wedge \\ \quad (\pi_{i-1}\pi_i \notin L_p \wedge (\pi_{i-1}, \pi_i) \notin [L_p]_{fn}) \implies \varsigma \in \mathcal{O}_f)\end{array}$$

Finally, based on **DC**, the dynamic characterization of context-updates, the definition of the property φ identifies separator instructions ς into the sets $\mathcal{O}_f$ (i.e., the separators that delimitate words pertaining to different functions in L_p) and $\mathcal{O}_g$ (i.e., the separators that delimitate words pertaining to the same function). The remaining operators are residual, part of $\mathcal{O}_r$, and are obtained in $\mathcal{O}$ due to the overapproximating character of the context-updates synthesis. Note that the residues $\mathcal{O}_r$ are constructs that may execute in programs' sequential order given by L_p; the gotos $\mathcal{O}_g$ and function calls $\mathcal{O}_f$ are constructs that break the sequential order for either jumping inside the current function body, or to another function, respectively. If the sets $\mathcal{O}_r$, $\mathcal{O}_g$, and $\mathcal{O}_f$ do not form a partition we use the remaining elements in $\mathcal{O}$ to signal counterexamples for the context-updates inference phase.

Next we describe the sets TP of benchmark tests, the function $TestTech$ we used for the experimental semantics WhileFun and MIPS.

6.1 Experiments in Chisel

We evaluate our technique for dynamic characterization of context-updates on a set of programs TP provided by a real-time systems benchmark called PapaBench [15]. The $TestTech$ function used to generate input test is based on a path coverage testing provided by the tool PathCrawler [12]. Next, we present the PapaBench benchmark forming the set TP, followed by the combined workflow of Chisel and PathCrawler where PathCrawler is used to generate the input states $In(p), \forall p \in TP$ and Chisel uses the Maude debugger to produce the execution traces $\Pi_p, \forall p \in TP$ which are later used with φ and L_p to refine the set of context-updates $\mathcal{O}$ obtained by **SC**.

PapaBench is extracted from a real-time design for an Unmanned Aerial Vehicle (UAV) application. The code presents the characteristics of an avionics application, and by extension, of a real-time design. First, it is *modular* at both structural and functional code levels; in this latter case there are several (exclusive) functional modes. Second, it contains a *global scheduling* to handle the high-level interleaving of the different functionalities. PapaBench has two communicating applications: a command management called `fly_by_wire` and a navigation management called `autopilot`. The functionalities are referred to as tasks and both these applications execute these tasks in control loops (i.e.,

Application fly_by_wire	LOC (WhileFun)	LOC (MIPS)	# Vars	# Test cases	#Total paths	Branch coverage
T1	119	534	19	54	75	95.45%
T2	59	329	21	11	14	100%
T3	82	501	26	37	40	92.86%
T4	50	235	15	60*	> 15*	>50%*
T5	66	453	16	27	27	92.86%
Application autopilot	**LOC (WhileFun)**	**LOC (MIPS)**	**# Vars**	**# Test cases**	**#Total paths**	**Branch coverage**
T6	306	1329	31	173	236	87.50%
T7	57	426	19	181	183	100%
T8	54	219	13	48	72	100%
T9	87	617	32	59	97	100%
T10	102	1002	25	60	90	95%
T11	15	90	10	11	11	100%
T12	49	363	23	78	102	100%
T13	240	1535	18	41	400	98.75%

Fig. 5. Refinement phase - testing coverage for PapaBench tasks using PathCrawler. *Test cases from multiple runs of PathCrawler.

the so-called *global schedulings*). The application `fly_by_wire` has the following five tasks: `T1` - receive_radio_commands, `T2` - send_data_to_autopilot, `T3` - receive_data_from_autopilot, `T4` - transmit_servos and `T5` - check_failsafe. The application `autopilot` has the following eight tasks: `T6` - manage_radio_commands, `T7` - control_stabilization, `T8` - send_data_to_fbw, `T9` - receive_gps_data, `T10` - control_navigation, `T11` - control_altitude, `T12` - control_climb and `T13` - manage_reporting. Also, each application serves three interrupts, which are not of concern in the overall evaluation.

Semantically, PapaBench features two interacting functionality modes: manual and automatic. In the manual mode, the execution of the radio command task `T1` triggers the task `T2` responsible with data transmission to the `autopilot` application. In turn, `autopilot` analyzes this data and responds to the `fly_by_wire` application (i.e., task `T8`) the necessary information on the radio commands, task `T6` and the flight stabilization, task `T7` for processing and issuing commands, in tasks `T3` and `T4`. The `autopilot` triggers the automatic mode when it receives GPS coordinates, task `T9` and enables navigation, altitude and climb control, tasks `T10`, `T11` and respectively `T12`. Finally, `fly_by_wire` handles failure checking with task `T5` and `autopilot` uses a parameter report manager, as task `T13`.

We conduct our experiments on the following settings: we run Chisel with Maude (and Full-Maude) 2.7 on a MacBook Pro 2.5 GHz, 4GB RAM, with PapaBench version 0.4 (for the WhileFun code) and the `gcc 4.7.1` cross-compiler to obtain MIPS code (and with sufficient traceability to check the corresponding program slices at the high- and low-levels). The results of the context-updates synthesis in Chisel are refined with the path-coverage testing of PathCrawler.

Briefly, the PathCrawler tool automatically generates test sets for a subset of C (and hence of our considered WhileFun language) with complete coverage of all feasible execution paths. The path-coverage strategy uses propagation of symbolic values coupled with constraint solving support.

Our context-updates refinement with PathCrawler generates test sets for imperative code and uses `gcc` without optimizations to obtain test sets for the binary code. We report the path-coverage strategy using PathCrawler on PapaBench, in Fig. 5. Whereas code compilation without optimizations does not guarantee the exact preservation of the high-level test statements, in this paper we assume an one-to-one mapping of tests. The last two columns of Fig. 5 (i.e., `Total paths` and `Branch coverage`) show the PathCrawler results on the total number of covered paths and the branch coverage factor for the imperative code. Under the previously mentioned assumption, we consider the same statistics to the MIPS code. The columns `Vars` and `Test cases` present the test size and respectively the necessary number of test cases to report path coverage. We use PathCrawler to generate 840 test sets for the 13 tasks of `autopilot` and `fly_by_wire`, with significant branch coverage for all but task `T4`. In this case, PathCrawler guarantees at least a 50% branch coverage (i.e., certain variables consider restricted domain values) but it fails to return a result for a less constraint variable set. We vary the domain values for several variables and collect multiple instances of PathCrawler on `T4` code.

The reduction factors obtained for the context-updates synthesis with path-coverage refinement are the same (and hence not reported again in Fig. 5) as in [4], whereas the path-coverage strategy is more powerful than the rule-coverage strategy. The first results [20] on PapaBench required manual annotation of the context-updates. In the current work and its previous draft [4] we automatically extract the context-updates set which is then refined with path-coverage and respectively rule-coverage testing. Because the formal language specifications are given as rewriting logic theories, we initially drafted, in [4], a testing methodology based on rule-coverage. As such, we randomly generated test cases in an attempt to cover a significant part of the program path (because the rule-coverage is agnostic to the program semantics). For particular programs and with carefully designed set of random test cases, it is possible to cover the exact set of language constructs, as we shown in [4]. But in general it is difficult to report coverage percentages in the rule-based testing. The current work with path-coverage testing reports, as for [4] the exact results for WhileFun while for MIPS the over-approximation at the synthesis phase is too large (the synthesized set of context-updates for MIPS includes most of the language instructions). Hence, the refinement phase, which is under-approximating the synthesized context-updates, is essential for context-updates synthesis in MIPS. As such, PathCrawler offers accurate path-coverage factors to exercise, in a systematic way, the most (if not all) context-updates constructs in the MIPS code.

7 Concluding remarks and future work

In this paper we have presented a generic synthesis method for context-updates synthesis, directly from formal language semantics written in Maude. The synthesis strategy performs a context-updates overapproximation, followed by an underapproximation refinement based on a path-coverage strategy (provided by a specialized tool – PathCrawler). We integrated our method in Chisel, a Maude tool that can perform generic program slicing. We experimented with imperative and assembly language semantics on a standard avionics application.

As ongoing work we focus on a more complex strategy for the refinement step by using more evolved testing strategies. For future work, we plan to extend the language with pointers, hence supporting more complex memory policies based on a more refined memory model. Finally, our aim is to introduce concurrency in the framework so that we can cover and test out proposed methodology on a larger and significant class of programming languages.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
2. M. Alpuente, D. Ballis, F. Frechina, and J. Sapina. Combining runtime checking and slicing to improve Maude error diagnosis. In *LRC*, pages 72–96, 2015.
3. I. M. Asavoae, M. Asavoae, and A. Riesco. Towards a formal semantics-based technique for interprocedural slicing. In *iFM 2014*, pages 291–306, 2014.
4. I. M. Asavoae, M. Asavoae, and A. Riesco. Context-updates analysis and refinement in Chisel. *CoRR*, abs/1709.06897, 2017.
5. D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In *FSE14*, pages 109–120, 2014.
6. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
7. J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *POPL*, pages 379–392. ACM Press, 1995.
8. N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.
9. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, 1990.
10. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, pages 35–46, 1988.
11. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
12. N. Kosmatov, W. Nicky, B. Botella, M. Roger, and O. Chebaro. A lesson on structural testing with pathcrawler-online.com. In *TAP 2012*, pages 169–175, 2012.
13. W. B. Langdon, S. Yoo, and M. Harman. Inferring automatic test oracles. In *ICSE*, pages 5–6, 2017.
14. J. Meseguer and G. Rosu. The rewriting logic semantics project. *TCS*, 373(3):213–237, 2007.
15. F. Nemer, H. Casse, P. Sainrat, J. P. Bahsoun, and M. D. Michiel. Papabench: a free real-time benchmark. In *WCET*, 2006.

16. T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: a calculus of context-dependent computation. In *ICFP*, pages 123–135, 2014.
17. A. Riesco. Using big-step and small-step semantics in Maude to perform declarative debugging. In *FLOPS*, pages 52–68, 2014.
18. A. Riesco, I. M. Asavoae, and M. Asavoae. A generic program slicing technique based on language definitions. In *WADT*, pages 248–264, 2013.
19. A. Riesco, I. M. Asavoae, and M. Asavoae. Memory policy analysis for semantics specifications in Maude. In *LOPSTR*, pages 293–310, 2015.
20. A. Riesco, I. M. Asavoae, and M. Asavoae. Slicing from formal semantics: Chisel. In *FASE*, pages 374–378, 2017.
21. G. Rosu. K - a semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*. IOS Press, 2017.
22. G. Rosu. Matching logic. *Logical Methods in Computer Science*, to appear, 2017.
23. G. Rosu and T. F. Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
24. M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978.
25. J. Talpin and P. Jouvelot. The type and effect discipline. In *LICS*, pages 162–173, 1992.
26. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
27. M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE Press, 1981.