A Generic Program Slicing Technique Based on Language Definitions*

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

[1] Universidad Complutense de Madrid, Spain
`ariesco@fdi.ucm.es`
[2] Alexandru Ioan Cuza University, Romania
{`mariuca.asavoae,mihail.asavoae`}`@info.uiac.ro`

Abstract. A formal executable semantics of a programming language has the necessary information to develop program debugging and reasoning techniques. In this paper we choose such a particular technique called program slicing and we introduce a generic algorithm which extracts a set of side-effects inducing constructs, directly from the formal executable semantics of a programming language. These constructs are further used to infer program slices, for given programs and specified slicing criteria. Our proposed approach improves on the parametrization of the language tools development because changes in the formal semantics are automatically carried out in the slicing procedure. We use the rewriting logic and the Maude system to implement a prototype and to test our technique.

Keywords: slicing, semantics, Maude, debugging.

1 Introduction

The intrinsic complexity of a modern software system imposes the need for specialized techniques and tool support, both targeting the system design and analysis aspects. It is often the case that these two aspects of the software development are inter-dependent. On the one hand, abstraction techniques are widely used solutions to reduce, in a systematic way, the size of the system under consideration. However this abstraction-based simplification process is usually dependent on quality and performance-driven refinements which, in turn, would benefit from tool support. On the other hand, the development of useful tool support requires sound techniques to ensure the correctness of the produced results. One possible solution to integrate techniques and tools development is to use a formal and executable framework such as rewriting logic [12]. Thus, a rewriting logic general methodology for design and analysis of complex software systems could and should rely on a formal executable programming language semantics to ground the development of both abstractions and tools.

* Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

A formal executable semantics of programming languages provides a rigorous mechanism to execute programs and in extenso, to implicitly or explicitly have access to all the program executions. The rewriting logic implementation—the Maude system [6]—comes with reachability and fully-fledged LTL model checking tool support. Thus, the notion of execution could be extended from program execution to analysis tool execution (over the same particular program). In these two settings, it is often important to simplify the executions with respect to certain criteria. One such simplification is called slicing [21] and, when applied on programs, it defines safe program fragments with respect to a specified set of variables.

In this paper we investigate, from a semantics-based perspective, the interdependent relationship between the program slicing general technique and its afferent tool—the program slicer. Modifications (i.e. extensions or abstractions) at the level of the programming language, and which are carried out at the level of the program, should be automatically reflected in the program slicing tool support. Therefore, we propose a static technique for program slicing which is based on a meta-level analysis of the formal executable semantics of programming languages. Our program slicing builds on the formal executable semantics of the language of interest, given as a rewriting logic theory, and on the source program.

Our procedure for program slicing consists of the following two steps: (1) a generic analysis of the formal executable semantics, followed by (2) a data dependency analysis of the input program. Step (1) is a fixpoint computation of the set of the smallest language constructs that may issue side-effects. Step (2) uses the resulting set from step (1) to extract safe program slices based on a specified set of variables of interest. Though our slicing technique is general, we exemplify it on the Maude specification of the classical WHILE language, named WhileL, augmented with a side-effect assignment and read/write statements. Next we present a quick, high-level overview of the semantics-based slicing methodology, grounded on the design of a formal executable semantics.

We use the standard approach to specify the WhileL programming language in rewriting logic. First, we define the (abstract) syntax, followed by the language configuration (i.e. the necessary semantic entities to define the behavior) and the language semantics (i.e. equations and rewrite rules).

We motivate our program slicing approach, starting with an alternative view on the language semantics. We elaborate on both structural and functional aspects of this view. Structurally, the formal definition of the WhileL language consists of three layers. At the top level there are the pure syntactic language constructs (i.e. the syntax), at the bottom there is the language state, while the middle layer contains the semantics equations and rewrite rules. This arrangement is important for the functionality of the definition. When we execute a program through these layers, its statements are decomposed into smaller syntactic constructs (i.e. found at the top layer) which, through transformations in the middle layer could result into state updates. In this way, a program execution establishes connections between the syntactic constructs and state updates, in other words which constructs yield side-effects. Step (1) of our program slicing method covers these meta-executions of the semantics to identify the set of syntactic constructs which results in state updates. This coverage employs unification [4] and an adaptation of the backward chaining technique [17]. During the language semantics analysis, the algorithm unfolds the middle layer (i.e. the semantics rewrite rules) into a special tree, applying labels to the visited nodes. This label-based classification is used to identify the set of side-effect constructs and to prune the unfolding tree. Step (2) takes the term representation of the input program together with the results from the previous step, represented as subterms, and does program slicing through term slicing.

Let us consider, throughout this paper, the WhileL program, say P, in Fig. 1 (left) on which we exemplify a standard slicing. We start with P and a set of variables of interest V. For example, V is {`p`} in Fig. 1 (middle) and V is {`s`} in the same figure (right). We identify and label the contexts containing variables of interest and add into the set V the other variables appearing in the current context. We run this step until V stabilizes. At the end, the program slice is represented by the skeleton term containing all the labeled contexts. Computing the slice of P with respect to variable `p`, in Fig. 1 (middle), identifies in the first iteration, the two assignment instructions to `p`. The second assignment adds the variable `i` to the V and the second iteration of the algorithm identifies the three input/output instructions. A third iteration considers the variable `j`, which contributes to the partially computed slice with its corresponding read instruction.

```
 read i ; read j ;          read i ; read j ;     read i ; read j ;
 s := 0 ; p := 1 ;          p := 1 ;              s := 0 ;
 while not (i == 0) do {
      write (i - j) ;       write (i - j) ;       write (i - j) ;
      s := s + i ;                                s := s + i ;
      p := p * i ;          p := p * i ;
      read i ;              read i ;              read i ;
}
```

Fig. 1. A WhileL program (left) and program slices, w.r.t. variable `p` (middle) and w.r.t. variable `s` (right)

The rewriting logic definitions of programming languages support program executability, and at the same time, provide all the necessary information to build analysis tools. In this paper we propose a generic algorithm based on a meta-level analysis of the language semantics, which extracts useful information (i.e. side-effect constructs) for the program slicing procedure. The actual program slice computation is through term slicing.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 introduces Maude and presents an example that will be used throughout the rest of the paper. Section 4 describes the slicing algorithm and the main theoretical results, while Section 5 shows our Maude prototype of the technique and outlines its implementation. Finally, Section 6 concludes and presents some subjects of future work.

2 Related Work

Program slicing is a general and well-founded technique that has a wide range of applications in program parallelization [18], debugging [1,16], testing [9] and analysis [11,15]. It was introduced in [21] where, for a given program, is used to compute executable fragments of programs, statically (i.e. without taking into consideration the program input). Strictly from the computation perspective of program slices, the work in [21] produces backward slices, while our approach applies the set of side-effect language constructs, obtained from the formal semantics, to produce forward slices. Informally, this kind of slices, introduced in [10], represents program fragments which are affected by a particular program statement, with respect to a set of variables of interest. With respect to the general problem of program slicing, we refer the reader to the work in [19] for a comprehensive survey on program slicing techniques, with an emphasis on the distinctions between forward and backward slicing approaches and between static and dynamic slicing methods.

Our proposed approach relies on a formal executable semantics definition, specified as a rewriting logic theory, over which we apply semantics-based reasoning techniques to extract side-effect language constructs. Next we elaborate on the works in [8,2,7], all of which use formal language definitions as support for developing program slicing methods.

The approach in [8] applies slicing to languages specified as unconditional term rewriting systems. It relates the dynamic dependences tracking with reduction sequences and, applying successive transformations on the original language semantics, it gathers the necessary dependency relations via rewriting. The resulting slice is defined as a context contained in the initial term representation of the program, a context being a subset of connected subterms. Our approach handles the formal semantics at a meta-level, without executing its rewrite rules.

The recent work in [2] proposes a first slicing technique of rewriting logic computations. It takes as input an execution trace—the result of executing Maude model checker tools—and computes dependency relations using a backward tracing mechanism. Both this work and its sequent extension to conditional term rewriting systems, in [3], perform dynamic slicing by executing the semantics for an initial given state. In comparison, we propose a static approach that is centered around the rewriting logic theory of the language definition. Moreover, our main target application is not counterexamples or execution traces of model checkers, but programs executed by the particular semantics.

Our two step slicing algorithm resembles the approach in [7], where an algorithm mechanically extracts slices from an intermediate representation of the language semantics definition. The algorithm relies on a well-defined transformation between a programming language semantics and this common representation. This transformation is non-trivial and language dependent. The approach in [7] also generalizes the notions of static and dynamic slices to that of constrained slices. What we propose is to eliminate the translation step to the intermediate representation and to work directly on the language semantics.

Finally, the work in [14] presents an approach to generate test cases similar to the one presented here in the sense that both use the semantics of programming languages formally specified to extract information about programs written in this languages. In this case, the semantic rules are used to instantiate the state of the variables used by the given program by using narrowing; in this way, it is possible to compute the values of the variables required to traverse all the statements in the program, the so called coverage.

3 Preliminaries

We present in this section the Maude system [6] by means of an example.

3.1 Maude

Maude modules are executable rewriting logic specifications. Rewriting logic [13] is a logic of change very suitable for the specification of concurrent systems. It is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [5], which, in addition to equations, allows one to state membership axioms characterizing the elements of a sort. Rewriting logic extends *MEL* by adding rewrite rules.

Maude functional modules [6, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Maude system modules [6, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

We present Maude syntax for functional modules specifying the natural numbers in the `MY-NAT` module. It defines the sorts `NzNat` and `Nat`, stating by means of a subsort declaration that any term with sort `NzNat` also has sort `Nat`:

```
fmod MY-NAT is
 sort NzNat Nat .     subsort NzNat < Nat .
```

The constructors for terms of these sorts, the constant `0` and the successor operator, are defined as follows:

```
 op 0 : -> Nat [ctor] .          op s : Nat -> NzNat [ctor] .
```

We can also define addition between natural numbers. First, we define the operator `_+_`, where the underscores are placeholders and that has attributes stating that it is commutative and associative. Then we specify its behavior by means of equations. These equations can be conditional, as shown in `add1`, where we check that the first argument is `0`, and can use patterns on the left-hand side, as shown in `add2`:

```
  vars N M : Nat .

  op _+_ : Nat Nat -> Nat [comm assoc] .
  ceq [add1] : N + M = N if N == 0 .
  eq [add2] : s(N) + M = s(N + M) .
 endfm
```

The syntax for system modules is presented together with the semantics of the WhileL language, that we will use for our slicing example. For this semantics, assume we have defined in `EVALUATION-EXP-EVAL` the syntax of a language with the empty instruction `skip`, assignment, assignment with addition (`_+=_`), increment operator (`_++`), `If` statement, `While` loop, composition of instructions, and `Read` and `Write` functions via a read/write buffer, all of them of sort `Com`;[1] some simple operations over expressions and Boolean expressions such as addition and equality; and a state, of sort `ST`, mapping variables to values. Using this module we specify the evaluation semantics of this language in `EVALUATION-SEMANTICS`, that first defines a `Program` as a triple of a term of sort `Com`, a state for the variables, and a state for the read/write buffer:

```
(mod EVALUATION-SEMANTICS is
  pr EVALUATION-EXP-EVAL .
  op <_,_,_> : Com ENV RWBUF -> Statement .
```

The rule `AsR` is in charge of the semantics of the assignment. It first evaluates the assigned expression (note that we use another operator `<_,_>` to evaluate expressions which does not require the read/write buffer) obtaining its value `v` and a new state for variables `st'`, and then updates this new state by introducing the new value for the variable:

```
  crl [AsR] : < X := e, st, rwb > => < skip, st'[v / X], rwb >
   if < e, st > => < v, st' > .
```

This update is in charge of the `upd` equation, that removes the variable from the state (if the variable is not in the state it remains unchanged) with the `remove` function and then introduces the new value:

```
  eq [upd] : ro [V / X] = remove(ro, X) X = V .
```

The rule `Inc1` also uses this update operator to increase the value of `X` in the state. The new value is computed by first obtaining the value `v` of `X` and then adding `1` using the auxiliary `Ap` function, that applies the given operation (addition in this case) to the values:

[1] The assignment with addition, the increment operator, and the read/write buffer extend the specification of the language given in [20].

```
crl [Inc1] : < X ++, st, rwb > => < skip, st[ Ap(+., v, 1) / X ], rwb >
 if < X, st > => < v, st > .
```

Similarly, the rule `SdE` describes the behavior of the `+=` assignment. It first computes the value of `X` in the given state to obtain the value `v` and then evaluates the expression `e` to obtain its final value `v'`; these values are added with `Ap`:

```
crl [SdE] : < X += e, st, rwb > => < skip, st''[ Ap(+.,v,v') / X ], rwb >
 if < X, st > => < v, st' > /\
    < e, st' > => < v', st'' > .
```

The rule `WriteR` introduces a new value in the read/write buffer after evaluating it:

```
crl [WriteR] : < Write e, st, rwb > => < skip, st', insert(v, rwb) >
 if < e, st > => < v, st' > .
```

where `insert` just introduces the value at the end of the buffer. Reading is performed by using the rule `ReadR1`. It tries to extract the next value from the buffer and, if it is not the `err` value, updates the state with it:

```
crl [ReadR1] : < Read X, st, rwb > => < skip, st[v / X], rwb' >
 if (v, rwb') := extract(rwb) /\
    v =/= err .
```

4 Semantics Based Program Slicing

In this section we discuss the semantics-based program slicing. This approach consists of two steps, namely the language semantics specification analysis and the program slicing as term rewriting. We introduce the algorithm for the semantics based analysis and present its execution on the WhileL language case study. We end this section with the results of the second step of the program slicing algorithm, applied on the example program in Fig.1 (left).

We consider $\mathcal{S}$ a specification of a program language semantics given in rewriting logic where we make the distinction between the languages syntax and the program state, via their different sorts in $\mathcal{S}$. For example in the semantics of the WhileL language described in Section 3 the language syntax is given by the sort `Com`, while the program state is formed by sorts `ENV` and `RWBUF`.

We consider the following (standard) transformations over $\mathcal{S}$. First, all equations are directed from left to right such that they become rules. Also, matching in conditions $v := e$ become $e \Rightarrow v$. Then, we assume a unique label R identifying each rule as follows:

$$[R] \ : \ l \Rightarrow r \text{ if } l_1 \Rightarrow r_1 \wedge \ldots \wedge l_n \Rightarrow r_n$$

with $n \in \mathbb{N}$. Finally, we transform any rule $[R]$ into a labeled Horn clause:

$$[R] \ l :- \ l_1; \ldots; l_n; r.$$

We denote $\bar{\mathcal{S}}$ the specification $\mathcal{S}$ after these transformations. Note that $\bar{\mathcal{S}}$ does not contain the r_i terms of the rules in $\mathcal{S}$. We comment on this later when describing the first slicing step algorithm.

Let t be a term with variables. We use the notation $R :: t$ for "an instance of t on non-variable position can be reduced by the rule R", i.e., exists an unifier θ and a subterm s of t such that $s\theta = l\theta$.

Definition 1. *A* hypernode *for a valid term t, denoted as* $\boxed{\forall R \in \bar{\mathcal{S}}, R :: t}$, *is a list* $\boxed{R_1 \to \ldots \to R_m}$ *of distinct rules in $\bar{\mathcal{S}}$, with $m \in \mathbb{N}$, such that $R_i :: t$ for all $1 \leq i \leq m$. We define an* inspection tree $i\mathcal{T}$ *as a tree of hypernodes where the children of a hypernode* $\boxed{R_1 \to \ldots \to R_m}$ *are defined as:*

$$children(\boxed{R_1 \to \ldots \to R_m}) = \{successors(R_i) \mid 1 \leq i \leq m\}$$

where the successors *of a Horn clause* $[R]\ l :- t_1; \ldots ; t_n.$ *are defined as:*

$$successors(R) = \boxed{\forall R_1 \in \bar{\mathcal{S}}, R_1 :: t_1} \longrightarrow \ldots \longrightarrow \boxed{\forall R_n \in \bar{\mathcal{S}}, R_n :: t_n}$$

Example 1. We give next a generic example of an inspection tree, considering $R_1, \ldots, R_7$ as all rule labels in $\bar{\mathcal{S}}$.

$$\begin{tikzcd}
 & \boxed{R_1 \to R_2} \arrow[dl] \arrow[dr] & \\
\boxed{R_3 \to R_4} \arrow[r] & \boxed{R_5} & \boxed{R_6 \to R_7}
\end{tikzcd}$$

The shape of this tree is induced by the rules $R_1, \ldots, R_7$ as follows: assume there exist t and two unifiers θ_1 and θ_2 such that $t\theta_1 = l_{R_1}\theta_1$ and $t\theta_2 = l_{R_2}\theta_2$, and that no other rule in $\bar{\mathcal{S}}$ can reduce t. Hence, $\boxed{\forall R \in \bar{\mathcal{S}}, R :: t}$ is $\boxed{R_1 \to R_2}$. Moreover, assume that $[R_1]\ l_{R_1} :- t_{1,1}; t_{2,1}.$ and $[R_2]\ l_{R_2} :- t_{1,2}.$ and assume there exist θ_i, with $3 \leq i \leq 7$ such that $t_{1,1}\theta_3 = l_{R_3}\theta_3$ and $t_{1,1}\theta_4 = l_{R_4}\theta_4$, $t_{2,1}\theta_5 = l_{R_5}\theta_5$, while $t_{1,2}\theta_6 = l_{R_6}\theta_6$ and $t_{1,2}\theta_7 = l_{R_7}\theta_7$. Hence, the down arrows starting in the rules R_1 and R_2 point towards $successors(R_1)$ and $successors(R_2)$, respectively (again, under the assumption that no other rules in $\bar{\mathcal{S}}$ can reduce $t_{1,1}, t_{1,2}$, or $t_{2,1}$). Namely, $\boxed{\forall R \in \bar{\mathcal{S}}, R :: t_{1,1}}$ is $\boxed{R_3 \to R_4}$, $\boxed{\forall R \in \bar{\mathcal{S}}, R :: t_{2,1}}$ is $\boxed{R_5}$, and $\boxed{\forall R \in \bar{\mathcal{S}}, R :: t_{1,2}}$ is $\boxed{R_6 \to R_7}$. Finally, the tree is completely unfolded under the assumption that $successors(R_i) = \emptyset, 3 \leq i \leq 7$.

The algorithm in Fig. 2 computes the set of basic syntactic language constructs which *may* produce side-effects, by inspecting the conditions and the right-hand side of each rewrite rule in the definition. For this inspection, we rely on unification [4] and a backward chaining technique [17]. The algorithm unfolds the semantics rewrite rules into the inspection tree $i\mathcal{T}$ such that the final tree contains a superset of all rules which can be called during a rewrite execution "`rew`

```
Input:  The language specification $\bar{S}$, the valid term $runPgm(X : LS, Y : PS)$,
        and the set $A$ of side-effect-sources (pre-computed based on $PS$).
Output: The basic syntactic constructs (non-recursive operators of sort $LS$)
        which induce side effects in the program state (of sorts $PS$).

$maySE$:=$A$;  $noSE$:=$\emptyset$;  $rMix$:=$\emptyset$;  $lBorder$:=empty stack;
$i\mathcal{T} := \boxed{\forall R \in \bar{\mathcal{S}}, R :: runPgm(X : LS, Y : PS)}$;
$lBorder$:=$insert$(select $R$ from $\forall R \in \bar{\mathcal{S}}, R :: runPgm(X : LS, Y : PS)$);

while $lBorder \neq$ empty stack do
   $R_{curr}$:=$top(lBorder)$;
   if $R_{curr} \in maySE \cup noSE$
   then $backtrack(i\mathcal{T}, lBorder)$;
   else if $R_{curr} \in lBorder - top(lBorder)$ or $R_{curr} \in rMix$
   then $rMix$+=$R_{curr}$; $backtrack(i\mathcal{T}, lBorder)$;
   else if $newSuccessors(R_{curr}) \neq \emptyset$
   then $i\mathcal{T}$+=$successors(R_{curr})$; $lBorder$:=$insert$(select $R$ from $successors(R_{curr})$);
   else if $successors(R_{curr}) \cap maySE \neq \emptyset$
   then $maySE$+=$R_{curr}$; $backtrack(i\mathcal{T}, lBorder)$;
   else $noSE$+=$R_{curr}$; $backtrack(i\mathcal{T}, lBorder)$;
od
return $\{s \in LS \mid s$ subterm of $l, [R]\ l{:-}t_1; \ldots ; t_n, R \in root(i\mathcal{T}) \cap (maySE - rMix)\}$
```

Fig. 2. The algorithm for detecting basic program syntax producing side effects

`rP`", where `rP` is a ground term with *runPgm* as top operator. More to the point, we assume that there exists in $\mathcal{S}$ an operator *runPgm* which is used for executing programs based on the language semantics specification $\mathcal{S}$. Note that usually *runPgm* contains as arguments the program term and the initial program state, i.e., *runPgm* is defined over *LS* (language syntax) and *PS* (program state) sorts. In other words, the inspection tree is an over-approximated result of a *rule reachability problem.*

We call t a *valid term* if its subterms and its variables meet a set of constraints *Cns* w.r.t. the possible ground instances of t. Typically we use a valid program term, formed only with the language syntax operators from *LS*, and valid program state term, formed only with the constructors of the sort *PS*. Note that for a valid term t some unifications are refuted, hence some rules are deleted from $\boxed{\forall R \in \bar{\mathcal{S}}, R :: t}$. The root of $i\mathcal{T}$ is the hypernode obtained from the input valid term $runPgm(X : LS, Y : PS)$. The rules in $i\mathcal{T}$'s hypernodes are either *Labeled* or *Unlabeled*, where $Labeled = lBorder \cup maySE \cup noSE \cup rMix$. *lBorder* is a stack which maintains the path currently unfolded in the inspection tree. *maySE* and *noSE* are two disjoint sets of rules which cover the *already traversed* side of the inspection tree (i.e., the rules in the left half-plane determined by the *lBorder*, *labeled border*, path). The *maySE* set contains rules which *may contain side-effects*, i.e., at least a rewrite starting in an *maySE* rule will reach the application of a rule which modifies the state of the program. We label as *noSE* the rules for which there is no side-effect propagation, hence $maySE \cap noSE = \emptyset$. The *rMix* label is introduced for detecting and pruning recursive rules, i.e., rules that may (indirectly) call themselves which means that at some point they appear twice in the *lBorder* stack.

The $backtrack(i\mathcal{T}, lBorder)$ subroutine is an augmented backtracking procedure (that goes up in *lBorder* as long as it cannot go right via the horizontal arrows in $i\mathcal{T}$). The augmentation consists in the fact that when *backtrack* removes the top of the *lBorder* stack, it also labels that rule as *maySE* or *noSE*. Namely, a rule is labeled *maySE* if, when removed from the top of the stack, it contains at least a successor rule labeled *maySE*. Otherwise, i.e., when no successor rule is labeled *maySE*, the rule is labeled *noSE* upon removal from the stack. The labeling in the *backtrack* subroutine induces the fact that at the end of the algorithm $rMix \subset maySE \cup noSE$. Also, $newSuccessors(R) = children(R) \backslash Labeled$. Finally, the algorithm returns the language syntax subterms of the terms l in the rules R from $i\mathcal{T}$'s root that are in *maySE* but not *rMix*. In other words, the result of the algorithm identifies the language syntactic constructs (subterms of sort LS) that are basic and may produce side-effects, i.e., the rules giving its semantics in $\mathcal{S}$ are non-recursive (not in *rMix*) and in *maySE*.

Our algorithm allows label propagation under certain conditions, with a labeled node summarizing the information of its corresponding subtree. The results of the language semantics analysis are used as contexts in the second slicing step to infer a safe program slice. Note that the termination of the algorithm in Fig. 2 is ensured by the fact that the specification has a finite number of rules, and that any rule in $i\mathcal{T}$ that was already *Labeled* is not unfolded anymore. Notice also that the algorithm is independent of the order of the labels in a hypernode. In fact, we consider the labels as a list just for ease the presentation, but hypernodes can be just considered sets of labels, since the algorithm relies on the *existence* of a rule label generating side effects to work.

Finally, recall that upon transformation of $\mathcal{S}$ to $\bar{\mathcal{S}}$ we do not consider the right hand-side of the rules' conditions (i.e., the terms r_i in the rule R have disappeared from $\bar{\mathcal{S}}$). We allow this because the algorithm in Fig. 2 computes an *over-approximation* of the rule reachability problem. Hence, by not considering the terms r_i in the semantic rules we include in the inspection tree a superset of the reachable rules. As another consequence, the *maySE* set is also over-approximated namely, we infer more potential side-effect syntactic constructs. However, observe that the actual side-effect rules are guaranteed to appear in the *maySE* set, again, because the algorithm relies on the *existence* of a *maySE* rule descendant to induce the side-effect character on a parent rule. Nevertheless, we could consider the terms r_i as well by augmenting the algorithm with additional pruning of the inspection tree. We leave this augmentation as future work.

Example 2. We exemplify the running of the algorithm in Fig. 2 over the semantics of the WhileL language, with the set of side effect sources formed by the rules for the operators `remove`, `insert`, and `extract`. We recall the definition of these operators:

```
op remove : ENV Variable -> ENV .
op insert : Value RWBUF -> RWBUF .
op extract : RWBUF -> PairValueRWBUF .

eq [rmv1] : remove(mt, X) = mt .
eq [rmv2] : remove(X = V ro, X')
          = if X == X' then ro else X = V remove(ro, X') fi .
eq [ins] : insert(V, buf) = buf V .
eq [ex1] : extract(nb) = err .
eq [ex2] : extract(V buf) = (V, buf) .
```

The set A of rules that produce modifications to the program state is formed by the rules labeled `rmv2`, `ins`, `ex2`, which are the only statements (either equations or rules) that modify the `ENV` or the `RWBUF` without using auxiliary functions. Also, `Com` is the language syntax sort LS, `ENV` and `RWBUF` give the program state sorts PS, while the "root" term is given by the operator `op <_,_,_> : Com ENV RWBUF -> Statement .`, i.e., *runPgm* is `< C:Com, E:ENV, B:RWBUF >`.

So, after the initial assignments before the loop, the variables in the algorithm are assigned as follows:
$\mathit{maySE} = \{\texttt{rmv2}, \texttt{ins}, \texttt{ex2}\}$, $\mathit{noSE} = \emptyset$, $\mathit{rMix} = \emptyset$,

$$
\begin{aligned}
i\mathcal{T} &= \boxed{\forall R \in \bar{\mathcal{S}},\ R ::\ \texttt{< C , E , B >}} \\
&= \boxed{\texttt{AsR} \rightarrow \texttt{Inc1} \rightarrow \texttt{Inc2} \rightarrow \texttt{SdE} \rightarrow \texttt{IfR1} \rightarrow \ldots \rightarrow \texttt{WriteR} \rightarrow \texttt{ReadR1} \rightarrow \texttt{ReadR2}}
\end{aligned}
$$

$\mathit{lBorder} = \texttt{AsR}$.

During the first iteration of the loop, the variable R_{curr} is `AsR`. We recall that `AsR` represents the following Horn clause:

```
[AsR]  < X := e, st, rwb > :- < e, st > ; < skip, st'[v / X], rwb > .
```

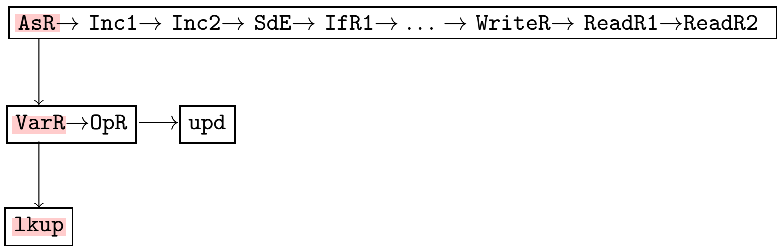
So l_1 is the term `< e, st >`, with variables `e:Exp` and `st:ENV`, while r is the term `< skip, st'[v / X], rwb >`, with variables `st':ENV`, `X:Var`, `v:Num`, and `rwb:RWBUF`. Consequently, *successors*(`AsR`) is given by the following list of hypernodes $\boxed{\forall R \in \bar{\mathcal{S}}, R :: l_1} \longrightarrow \boxed{\forall R \in \bar{\mathcal{S}}, R :: r}$, which evaluates to the following hypernodes: $\boxed{\texttt{VarR} \rightarrow \texttt{OpR}}$, and respectively $\boxed{\texttt{upd}}$. Note that the rule `rmv2` cannot be applied over the subterm `st` of l_1 because `st` is a valid term, formed only with the constructors of `ENV`. Hence, the loop executes the branch with the condition $\mathit{newSuccessors}(R_{curr}) \neq \emptyset$, and the iteration tree $i\mathcal{T}$ becomes:

$$\begin{tikzcd}
\boxed{\texttt{AsR} \rightarrow \texttt{Inc1} \rightarrow \texttt{Inc2} \rightarrow \texttt{SdE} \rightarrow \texttt{IfR1} \rightarrow \ldots \rightarrow \texttt{WriteR} \rightarrow \texttt{ReadR1} \rightarrow \texttt{ReadR2}} \arrow[d] \\
\boxed{\texttt{VarR} \rightarrow \texttt{OpR}} \arrow[r] & \boxed{\texttt{upd}}
\end{tikzcd}$$

Note that the color code in $i\mathcal{T}$ signifies that *lBorder* is the stack `VarR`, `AsR` (i.e., red for *lBorder*). The second iteration of the loop makes R_{curr} the Horn clause:

```
[VarR]  < X, st > :-  < st(X), st > .
```

So the same branch of the loop as before is executed and $i\mathcal{T}$ becomes:



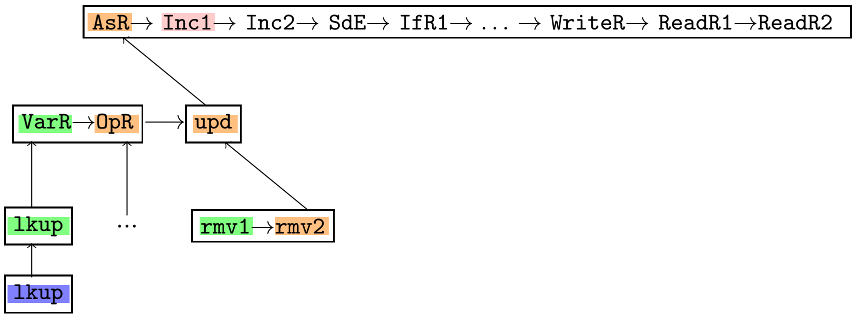
The next iteration of the loop finds that `lkup` is recursive, where

```
[lkup]  (X = V ro)(X') :- if X == X' then V else ro(X') fi .
```

because `lkup` $\in$ *successors*(`lkup`) = $\boxed{\texttt{lkup}}$ (as `ro(X')` unifies with `lkup`), so *lBorder* becomes the stack `lkup`, `lkup`, `VarR`, `AsR`. Hence, in the following iteration of the loop, the top `lkup` becomes an element of *rMix* and, upon backtracking, the next `lkup` becomes an element of *noSE* (since it does not have a *maySE* successor rule). For the same reason *backtrack*($i\mathcal{T}$, `VarR`, `AsR`) makes `VarR` an element of *noSE*, and *lBorder* becomes `OpR`, `AsR`.

The next few iterations of the loop make `OpR` an element of both *rMix* and *noSE* in the same way as `lkup`. We skip over these steps and explain from the iteration with the *lBorder* stack containing `upd`, `AsR`.

Since *successors*(`upd`)=$\boxed{\texttt{rmv1}\rightarrow\texttt{rmv2}}$ which is a hypernode that contains a *maySE* rule, i.e., `rmv2`, then the rule `upd` becomes an element of *maySE*, via the last branch in the algorithm. Also, during *backtrack*($i\mathcal{T}$, `upd`, `AsR`) also `AsR` becomes an element of *maySE*, because of its *maySE* successor rule in the hypernode $\boxed{\texttt{upd}}$. After these steps, the inspection tree has the following structure:



After the algorithm in Fig. 2 is applied to the WhileL language, we identify the following set of syntactic constructs that may produce side-effects: the assignment statement `_:=_`, the input/output statements `Read_` and `Write_` and the two special addition-based statements `_++` and `_+=_`. Note that if we consider only the program environment as the side effect source (i.e., the set A is $\{$`rmv2`$\}$) the algorithm produces all the previous side-effect syntactic constructs, besides `Write_`. The reason is that `WriteR` rule was previously labeled as *maySE* only due to the `ins` rule that appeared among its descendants. Since now $A = \{$`rmv2`$\}$, then `ins` is not *maySE*, so `WriteR` is labeled *noSE*.

The set of basic side-effect syntactic constructs together with the term representation of the program and the slicing criterion define the input state for the second step of our slicing algorithm which computes a slice of the program.

Definition 2. *We say that a program term p* produces side-effects over a vari*able v w.r.t. a side-effect set SE if the top operator of p is in SE and the variable v is a designated subterm of p. We define a* valid slice *of a program w.r.t. a slicing criterion V (i.e., a set of program variables) as a superset of program subterms that (indirectly) produce side-effects over some variables in V.*

The second slicing step is a fixpoint iteration which applies the *current* slicing criterion over the program term in order to discover new subterms of the program that use the slicing criterion, i.e., the program subterm produces side-effects over *some* variables in the slicing criterion. When a new subterm is discovered, the slicing criterion is updated by adding the variables producing the side-effects (e.g., the variables in the second argument of `_:=_`). We iterate this until the slicing criterion remains unchanged, so no new subterms can be discovered and added to the result, i.e., the slice set.

For example, the iterations of the second slicing step for the program in Fig. 1, the side-effect constructs obtained in Example 2, and the slicing criterion $\{$`p`$\}$ are listed in Fig. 3. Namely, the set $\{$`p`$\}$ is applied on the set of side-effect syntactic constructs to produce the slice subterms used on the term program: `p:=_, Read p, Write p, p++, p+=_`. Only the subterm `p:=_` is matched in the program, on the term assignment for `p`. This iteration results are shown in the first row, in Fig. 3. Because of the matched assignment `p := p *. i`, the slicing criterion becomes $\{$`p`, `i`$\}$. Under the new slicing criterion, the set of side-effect syntactic constructs is $\{$`i:=_, Read i, Write i, i++, i+=_`$\}$. This time, the term slicing of the program matches the two `Read i` instructions and the `Write (i-.j)` (the second row in Fig. 3). The algorithm reaches the fixpoint when the slicing criterion is the set $\{$`p`, `i`, `j`$\}$. The final iteration is graphically represented in Fig. 4, where the slice on the term program is identified based on the set of subterms inside surrounded area (inside a triangle or a diamond). Note that the upward triangles surround the subterms discovered at the first iteration, the downward triangle surrounds the second iteration new terms, while the diamond corresponds to the third iteration new terms. Also, note that the subterm `Read i` from the resulted slice set appears twice in the initial program.

The second slicing step algorithm terminates, because there exists a finite set of program subterms, and it produces a valid slice, because it exhaustively saturates the slicing criterion. Moreover, the result is a minimal slice w.r.t. the set of side-effect syntactic constructs given as argument. However, the obtained slice is not minimal mainly because the set of side-effect syntactic constructs obtained in the first slicing step is already an over-approximation.

Iteration	**Slicing variables**	**Computed slice** (identified subterms)
1 △	**p**	`p := 1, p := p *. i`
2 ▽	p, **i**	`Read i, p := 1, Write (i -. j), p := p *. i`
3 ◊	p, i, **j**	`Read i, Read j, p := 1, Write (i -. j), p := p *. i`
4 .	p, i, j	`Read i, Read j, p := 1, Write (i -. j), p := p *. i`

Fig. 3. Program slicing as term slicing - the fixpoint iterations

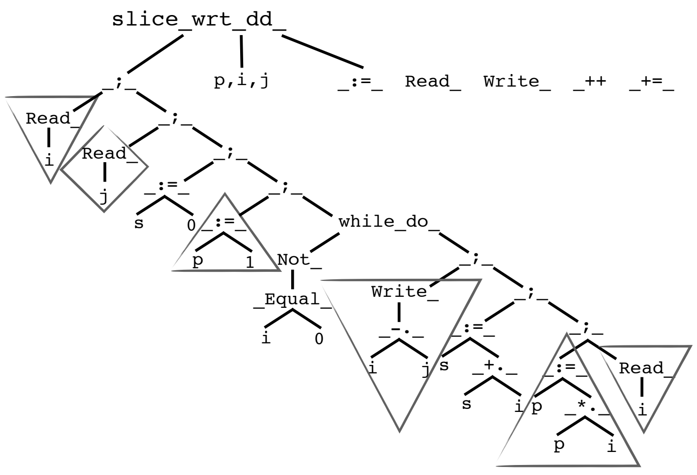


Fig. 4. Program slicing as term slicing—the result subterm

Based on the obtained slice set we can add structure to the slicing result by identifying the "skeleton" term that contains the slice set. For example, we can decide to keep the composed statements that contain subterms in the slice set. As such, the `while_do_` statement from the example program has subterms in the slice set (e.g., `Read i`) so we add it to the "structured" slice. The resulting program representing the "structured" slice is:

```
Read i;
Read j;
p := 1;
While _ do {
  Write(i -. j);
  p := p *. i;
  Read i; }
```

5 System Description

We present in this section our Maude prototype and some details about its implementation.

5.1 Slicing Session

The tool is started by loading the `slicing.maude` file available at `http://maude.sip.ucm.es/slicing`. It starts an input/output loop where modules and commands can be introduced. Once the module in Section 3 has been introduced, we have to introduce the sort of variables and the sorts where we want to detect side effects. With this information, the tool can compute the rules and the functions generating side effects:

```
Maude> (set variables sort Var .)
Var selected as sort for the variables.

Maude> (set side-effect sorts ENV RWBUF .)
ENV RWBUF selected as side effect sorts.
```

The slicing command follows the notation shown in the previous sections. The keyword `slice` is followed by the program we want to debug, the keyword `wrt` and the list of variables that will be used by the second step of the algorithm described in the previous section. The tool outputs the label of the rules inducing side effects and the name of the variables that have been computed during the slicing stage:

```
Maude> (slice (< 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,
               X:ENV, Y:RWBUF >) wrt p .)

The rules causing side effects are: AsR Inc1 ReadR1 SdE WriteR
The variables obtained by the slicing process are: p i j
```

5.2 Implementation Details

Exploiting the fact that rewriting logic is reflective, a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [6, Chapter 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. In this way, we can manipulate the modules introduced by the user, develop the slicing process, and implement the input/output interactions in Maude itself.

More specifically, our tool traverses the rules in the module indicated by the user to find the rules that modify the state of the terms modified by the side effects (`ENV` and `RWBUF` in our example). With these rules and using the predefined unification commands available in Maude our prototype can generate and traverse the hypernodes, thus computing the first step of the algorithm in the previous section. For the second part, it checks the left-hand sides of these rules, discarding the information of the side effects terms to focus on the instructions. It then uses this information to traverse the initial term (containing the program we are analyzing) checking the terms that appear in each of these terms. If any of the variables given as slicing criterion are used, then the rest of the variables appearing in this term are added to the current set and the process is repeated until the fixpoint is reached.

6 Concluding Remarks and Ongoing Work

We presented a two-phased technique for program slicing based on the formal executable semantics of a WHILE programming language, given as a rewriting logic theory. The first phase, called language semantics specification analysis considered an exhaustive inspection of the language semantics to extract a set of side-effect language constructs. The second phase was to perform program slicing as term slicing, using the previously computed set of primitives, the input (term representation of the) program and the slicing criterion. Both the formal definition of our language and the semantics-based slicing technique are implemented and tested in the Maude system.

We plan to extend this work on the following several directions. First, we incrementally analyze the impact on adding various side-effect constructs to the WhileL language, such as pointers, exceptions or file-manipulation operations. Second, we address other types of programming languages and their specific side-effect constructs. For example, if we consider assembly languages, it happens that various arithmetic instructions visibly modify a particular register value, and invisibly affect subsequent conditions in the program, via modifications of special arithmetic flags (i.e. overflow, sign, etc). Third, we improve the algorithm in the first step in our slicing technique in order to obtain a better (smaller) over-approximation of the produced set of side-effect syntactic constructs. We believe that pursuing these directions would further improve the current semantics-based program slicing technique and produce a useful design and analysis tool for language developers.

References

1. Agrawal, H., DeMillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and backtracking. Software - Practice and Experience 23(6), 589–616 (1993)
2. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward trace slicing for rewriting logic theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 34–48. Springer, Heidelberg (2011)

3. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Backward trace slicing for conditional rewrite theories. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 62–76. Springer, Heidelberg (2012)
4. Baader, F., Snyder, W.: Unification theory. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 445–532. Elsevier (2001)
5. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 379–392. ACM Press (1995)
8. Field, J., Tip, F.: Dynamic dependence in term rewriting systems and its application to program slicing. Information & Software Technology 40(11-12), 609–636 (1998)
9. Harman, M., Danicic, S.: Using program slicing to simplify testing. Journal of Software Testing, Verification and Reliability 5, 143–162 (1995)
10. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages Systems 12(1), 26–60 (1990)
11. Jhala, R., Majumdar, R.: Path slicing. In: Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 38–47. ACM Press (2005)
12. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theoretical Computer Science 285(2), 121–154 (2002)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
14. Riesco, A.: Using semantics specified in Maude to generate test cases. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 90–104. Springer, Heidelberg (2012)
15. Sandberg, C., Ermedahl, A., Gustafsson, J., Lisper, B.: Faster WCET flow analysis by program slicing. In: Proc. of the 2006 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2006, pp. 103–112. ACM Press (2006)
16. Silva, J., Chitil, O.: Combining algorithmic debugging and program slicing. In: Proc. of the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming, PPDP 2006, pp. 157–166. ACM Press (2006)
17. Sterling, L., Shapiro, E.Y.: The Art of Prolog - Advanced Programming Techniques. MIT Press (1986)
18. Tian, C., Feng, M., Gupta, R.: Speculative parallelization using state separation and multiple value prediction. In: Proc. of the 2010 International Symposium on Memory Management, ISMM 2010, pp. 63–72. ACM Press (2010)
19. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3), 121–189 (1995)
20. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67, 226–293 (2006)
21. Weiser, M.: Program slicing. In: Proc. of the 5th International Conference on Software Engineering, ICSE 1981, pp. 439–449. IEEE Press (1981)