Towards a Formal Semantics-Based Technique for Interprocedural Slicing[⋆]

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

[1] VERIMAG/UJF, France
{mariuca.asavoae,mihail.asavoae}@imag.fr
[2] Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Interprocedural slicing is a technique applied on programs with procedures which relies on how the information is passed at procedure call/return sites. Such a technique computes program slices (i.e. program fragments restricted w.r.t. a given criterion). The existing approaches to interprocedural slicing exploit the particularities of the underlying language semantics in order to compute program slices. In this paper we propose a generic technique for interprocedural slicing. More specifically, our approach works with inferred particularities of a language semantics, given as a rewriting-logic specification, and computes program slices using a term slicing-based algorithm.

Keywords: slicing, semantics, Maude, debugging.

1 Introduction

Complex software systems are built in a modular fashion, where modularity is implemented with functions and modules, in declarative-style programming; with classes and interfaces, in object-oriented programming; or with other means of organizing the code. Besides their structural characteristics, the modules also carry semantic information. The modules could be parameterized by types and values (e.g. the generic classes of Java and C#, the template classes of C++, or the parameterized modules of Maude and OCaml) or could have specialized usability (e.g. abstract classes in object-oriented languages).

It is preferable, for efficiency reasons, that the modular characteristics of a system are preserved when new analysis techniques and tool support are developed. One possible solution to integrate both analysis and tool development is to use a formal executable framework such as rewriting logic [13]. For any given program (correctly constructed w.r.t. the language syntax), the formal executable semantics, given as a rewriting logic specification, provides the set of all the concrete executions, for all the possible input data. Furthermore, the notion of a concrete execution extends to an abstract execution—as an execution with an analysis

⋆ This research has been partially supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04).

tool—and we have the set of concrete executions as the basis of any abstraction (and implicitly abstract execution) of a program. One particular abstraction is program slicing [26], which computes safe program fragments (also called slices) w.r.t. a specified set of variables. A complex variant of program slicing, called interprocedural slicing, preserves the modularity of the underlying program and exploits how the program data is passed between these modules.

Interprocedural slicing is the slicing method applied on programs with procedures where the slice is computed for the entire program, taking into account the procedure calls/returns. The main problem that arises in interprocedural slicing is related to the fact that the procedure calls/returns may be analyzed with a too coarse abstraction. Namely, the abstraction relies only on the call graph without taking into account the context changes (i.e., the instantiation of the local variables during a procedure execution) occurring during a procedure call/return. Since we develop a *generic*, formal semantics-based slicing method, we assume that we do not know which language constructs produce these context changes. Hence, we include in our slicing method a phase for inferring these constructs, denoted in the following as *scope-update constructs*.

Now, our proposed technique for interprocedural slicing has two phases which could be described as follows: Given a programming language semantics $\mathcal{S}$, in the first phase we extract scope-update constructs $\mathfrak{c}$ from $\mathcal{S}$ and, in the second phase, we use these constructs for the interprocedural slicing of $\mathcal{S}$-programs (i.e., programs written in the language specified by $\mathcal{S}$ which, in fact, are well-formed terms in $\mathcal{S}$). In this paper we focus on the second phase of the interprocedural program slicing, meaning the term slicing-based algorithm. The first phase, (i.e. the extraction of the scope-update constructs) follows a similar meta-analysis of the language semantics as in [17], where side-effect constructs are extracted. We require $\mathcal{S}$ to be expressed as a rewriting logic theory [13], which is executable and benefits of tool support via the Maude system [3], an implementation of the rewriting logic framework. The technique to obtain the scope-update constructs is, in fact, a meta-analysis of the programming language. The interprocedural program slicing uses $\mathfrak{c}$ to collect and propagate abstract information according to the scope switches from $\mathfrak{c}$. This technique is concretised with an implementation into a generic semantics-based slicing tool developed in Maude.

For presentation purposes, we consider a WHILE language [10] with functions and local variable declarations (which introduce variable scoping); we call this extension WhileF. Then, in order to differentiate two variables based on their scopes, we need to identify scope-update constructs at the level of the semantics. Note that the meta-analysis for scope-updates used in the present work is slightly more complicated than the one for side-effects described in [17], because scope-updates usually work in pairs so now we need to analyze $\mathcal{S}$ targeting pairs of operators (for procedure call and return). Such pairs could be explicitly presented in the language semantics definitions (through different rewrite rules for call and return) or implicitly, as in this work (with an explicit rewrite rule for call and implicit return instruction).

Another interesting difference consists in the fact that the second program slicing step receives as input both scope-update and side-effect information, which implies heavy changes. In this case, it is necessary to address the representability of the derived scope-update constructs w.r.t. the interprocedural program slicing. Namely, a combined representation of scope-update and side-effect constructs could consist in terms representing generic skeletons for procedure summaries (a succinct representation of the procedure behavior w.r.t. its input variables).

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 introduces the basic notions of program slicing and rewriting logic, used throughout the paper, while Section 4 describes the proposed program slicing as term slicing-based algorithm. Finally, Section 5 concludes and presents some lines of future work.

2 Related Work

Program slicing addresses a wide range of applications, from code parallelization [23] to program testing [8], debugging [21], and analysis [12,11]. Since our goal is to design and implement a semantics-based program analysis tool in a rewriting-logic environment, we relate our method to both interprocedural slicing in program analysis and in rewriting logic. With respect to the general problem of program slicing, we refer the reader to the comprehensive survey of slicing techniques, in [24].

The technique of program slicing was introduced in [26], and for a given program with procedures, it computes slices using a limited form of context information (i.e. before each procedure call). The approach resembles an on-demand procedure inlining, using a backward propagation mechanism (thus, producing backward slices). Our approach takes into consideration the context-update constructs (as extracted from the formal semantics) and produces forward slices (via term slicing on the term program). Moreover, the context-update constructs play the role of symbolic procedure summaries, as in [20,11,7,22]. A procedure summary is a compact representation of the procedure behavior, parameterized by its input values, which in our proposed framework is the context-update construct. The interprocedural slicing is explicit in [11,22] and implicit in [20,7], and sets the support for interprocedural program analyses.

The work in [20] uses a data-flow analysis to represent how the information is passed between procedure calls. It is applied on a restricted class of programs—restricted by a finite lattice of data values—, while the underlying program representation is a mix of control-flow and control-call graphs. In comparison, our approach considers richer context information (as in [11]), while working on a similar representation of a program (as a term). The work in [7] keeps the same working structures but addresses the main data limitation of [20]. As such, the procedure summaries are represented as sets of constraints on the input/output variables. The underlying interprocedural slicing algorithm of [7] is more refined than our approach (though not generic), just because of the richer representation of context information. We follow closely the work in [11], which introduces a new program representation for the interprocedural slicing. In comparison, our approach does not require the explicit context representation, but uses term matching to distinguish between different contexts.

In the rewriting logic environment, there are several approaches towards debugging [1], testing [16], and analysis [17]. The dynamic slicing technique in [1] works on execution traces of the Maude model checker. In comparison, we propose a static approach built around a formal semantics and with an emphasis on computing slices for programs and not for given traces (e.g. of model checker runs). The work in [16] 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 specific information. 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. The technique in the current paper follows our previous work on language-independent program slicing in rewriting logic environment [17]. Actually, the implementation of the current work is an extension of the slicing tool we developed in [17]. Both approaches share the methodology steps: (1) the initial meta-analysis of $\mathcal{S}$ and (2) the program analysis conducted over the $\mathcal{S}$-programs. More specifically, in [17] we use the classical WHILE language augmented with side-effect constructs (assignments and read/write statements) to exemplify (1) the inference of the set of side-effect language constructs in $\mathcal{S}$, and (2) the program slicing as term rewriting.

As a semantical framework, Maude has been used to specify the semantics of several languages, such as LOTOS [25], CCS [25], Java [6], or C [4]. These works describe a methodology to represent the semantics of programming languages in Maude, led to the *rewriting logic semantics project* [14] and to the development of the $\mathbb{K}$ [19] framework. We plan to use these semantics to perform program analysis in the future.

3 Preliminaries

Program slicing, as introduced in [26], is a program analysis technique which computes all the program statements that might affect the value of a variable v at a program point of interest, p. It is a common setting to consider p as the last instruction of a procedure or the entire program. Hence, without restricting the proposed methodology, here we consider slices of the entire program.

A classification of program slicing techniques identifies *intraprocedural* slicing when the method is applied on a procedure body and *interprocedural* slicing when the method is applied across procedure boundaries. The key element of a methodology for interprocedural slicing is the notion of context (i.e. the values of the function/procedure parameters). Next, we elaborate on how context-aware program slicing produces better program slices than a context-forgetful one.

Let us consider, in Fig. 1, the program from [11], written as an WhileF program term, upon which we present subtleties of interprocedural slicing. We start the slicing with the set of variables of interest $\{\mathtt{z}\}$.

The first method, in [26], resembles an on-demand inlining of the necessary procedures. In the example in Fig. 1, the variable {`z`} is an argument of procedure `Add` call in `Inc`, hence, the sliced body of `Add` is included in the slice of `Inc`. Note that, when slicing the body of `Add`, `z` is replaced by `a`. Hence, the slicing of `Add` deems {`a`} and {`b`} as relevant. The return statement of procedure `Inc` is paired with the call to `Inc`, in the body of `A` so the variable {`y`} becomes relevant for the computed slice. When the algorithm traces the source of the variable `y`, it finds the second call to `Add` in the body of `A` (with the arguments `x` and `y`) and includes it in the program slice. When tracing the source of `x` and `y`, it leads to include the entire body of procedure `Main` (through the variables `sum` and `i`, which are used by the assignments and calls of `Main`). Using this method, the program slice w.r.t. the set of variables of interest - {`z`}, is the original program, as in Fig. 1. This particular slice is a safe over-approximation of a more precise one (which we present next) because the method relies on a transitive-closure—fixpoint computation style where all the variables of interest are collected at the level of each procedure body. As such, the body of procedure `Add` is included twice in the computed slice.

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
   Call A (sum, i)
}

function A (x,y) {
  Call Add (x,y);
  Call Inc (y)
}
function Add (a,b) {
  a := a + b
}

function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Fig. 1. A WhileF program `Px` with procedures `Main`, `A`, `Add`, and `Inc`

The second approach in [11] exploits, for each procedure call, the available information w.r.t. the program variables passed as arguments (i.e. the existing context before the procedure call). Again, in the example in Fig. 1, the variable `z` is an argument of procedure `Add`. Hence, upon the return of `Add`, its body is included in the slice. However, because of the data dependencies between variables `a` and `b` (with `a` using an unmodified value of `b`) only the variable `a` is collected and further used in slicing. Next, upon the return statements of `Add` and then `Inc`, the call of `Inc` in `A` (with parameter `y`) is included in the slice. Note that the call to `Add` from `A` (with parameters `x` and `y`) is not included in the slice because it does not modify the context (i.e. the variables of interest at the call point in `A`). As such, the slicing algorithm collects only the second parameter of procedure `A`, and following the call to `A` in `Main`, it discovers `i` as the variable of interest (and not `sum` as it was the case of the previous method). Hence, the sliced `A` with only the second argument is included in the computed slice. Consequently, the variable `sum` from `Main` is left outside the slice. The result is presented in Fig. 2.

Any program analysis that computes an interprocedural slice works with the control-flow graph—which captures the program flow at the level of

```
function Main (){
 Local i;
 i := 1;
 while i < 11 do
   Call A (i)
}
```

```
function A (y) {
 Call Inc (y)
}
function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Fig. 2. The result of a context-dependent interprocedural analysis for `Px`

procedures—and the call graph—which represents the program flow between the different procedures—. To improve the precision of the computed program slice, it is necessary for the analysis to use explicit representations of procedure contexts (as special nodes and transitions). This is the case of the second method which relies on a program representation called *system dependence graph.*

4 Semantics-Based Interprocedural Slicing

We present in this section the algorithm for our interprocedural slicing approach, and illustrate it with an example. Then, we describe a Maude prototype executing the algorithm for semantics specified in Maude.

4.1 Program Slicing as Term Slicing

In [17] we described how to extract the set of side-effect instructions SE from the semantics specification $\mathcal{S}$ and how to use SE for an *intraprocedural slicing* method. In the current work we focus on describing the *interprocedural slicing method* which is built on top of the intraprocedural slicing result from [17].

The programs written in the programming language specified by $\mathcal{S}$ are denoted as $\mathfrak{p}$. By *program variables* we understand subterms of $\mathfrak{p}$ of sort Var. If we consider the *subterm* relation as $\preccurlyeq$, we have $v \preccurlyeq \mathfrak{p}$ where v is a program variable.

We consider a *slicing criterion sc* to be a subset of program variables which are of interest for the slice. We denote by SC the slicing criterion sc augmented with *data flow information* that is collected along the slicing method. Hence, SC is a set of pairs of program variables of form $\overset{\curvearrowleft}{v, v'}$, denoting that v depends on v', or just variables v, denoting that v is independent.

We assume as given the set of program functions $\mathfrak{F}_{\mathfrak{p}}$ defining the program $\mathfrak{p}$. We claim that $\mathfrak{F}_{\mathfrak{p}}$ can be inferred from the term $\mathfrak{p}$, given the $\mathcal{S}$-sorts defining functions, variables, and instruction sequences. We base this claim on the fact that $\mathfrak{p}$ is formed, in general, as a sequence of function definitions hence its sequence constructor can be automatically identified from $\mathcal{S}$. Also, we use getFnBody$(f, \mathfrak{F}_{\mathfrak{p}})$ to obtain the function identified by f in $\mathfrak{F}_{\mathfrak{p}}$. Note that getFnBody$(f, \mathfrak{F}_{\mathfrak{p}}) \preccurlyeq \mathfrak{F}_{\mathfrak{p}}$.

Furthermore, we denote the method computing the intraprocedural slicing as $\$(B, SC, SE)$, where B is the code, i.e., the body of some function f in $\mathfrak{p}$ (note that $B \preccurlyeq \mathfrak{p}$), while SC is a slicing criterion and SE is the set of side-effect constructs. Hence, $\$$ takes the body B of a function f and a slicing criterion

SC (i.e. a set of variables) and keeps only the parts of B that are subterms starting with a side-effect effect construct (from SE) and containing variables from the slicing criterion SC. The result of $\$(B, SC, SE)$ is given as a term $SC :: \mathit{fn}\langle \mathit{fn}(\mathit{fp}^\sharp)\{\mathit{fs}\}\rangle$ where SC is the data flow augmented slicing criterion, $\mathit{fn} \in \mathit{FunctionName}$ is a function identifier, and fs is the slice computed for fn. Meanwhile $\mathit{fp}^\sharp$ is the list of fn's formal parameters fp filtered by SC, i.e., all the formal parameters not appearing in SC are abstracted to a fixed additional variable $\sharp$.

Now we give a brief explanation on how the intraprocedural slicing \$ works. We say that a program subterm *modifies* a variable v if the top operator is in SE and v appears as a leaf in a specific part of the subterm, e.g., the variable v appears in the first argument of `_:=_` or in `Local_`. When such a subterm is discovered by \$ for a slicing variable then the slicing criterion is updated by adding the variables producing the side-effects (e.g. all variables v' in the second argument of `_:=_`) and the data flow relations $\overset{\curvearrowleft}{v, v'}$. We call fs a *skeleton subterm* of B and we denote this as $\mathit{fs} \precsim B$.

In Fig. 3 we give the slicing method, **termSlicing**, which receives as input the slicing criterion sc, the set of program functions $\mathfrak{F}_\mathfrak{p}$, and the set of side-effect and context-updates syntactic constructs, SE and CU, respectively. The output is the set of sliced function definitions *slicedFnSet* together with the obtained data flow augmented slicing criterion *dfsc*. Note that $\mathfrak{F}_\mathfrak{p}$, SE, and CU are assumed to be precomputed based on the programming language semantics specification $\mathcal{S}$. The algorithm for inferring SE is given in [17, Section 4]. The algorithm for inferring CU goes along the same lines as the one for SE and it is based on the automatic discovery of stack structures used in $\mathcal{S}$ for defining the programming language commands. For example, in WhileF the only command inducing context-updates is `Call_(_)` instruction. In the current work we assume CU given in order to focus on the interprocedural slicing as term slicing method. However, we claim that **termSlicing** is generic w.r.t. $\mathcal{S}$ since $\mathfrak{F}_\mathfrak{p}$, SE, and CU can be automatically derived from $\mathcal{S}$.

termSlicing is a fixpoint iteration which applies the *current* data-flow-augmented slicing criterion over the function terms in order to discover new skeleton subterms of the program that comply with the slicing criterion. The protocol of each iteration step is to take each currently sliced function and slice down and up in the *call graph*. In other words, the *intraprocedural slicing* is applied on every *called function* (i.e. goes *down* in the call graph) and every *calling function* (i.e. goes *up* in the call graph).

Technically, **termSlicing** relies on incrementally building the program slice, stored in *workingSet*, and the data flow augmented slicing criterion, stored in *dfsc*. This process has two phases: the initialization of *workingSet* and *dfsc* (lines 0-6) and the loop implementing the fixpoint (lines 7-39).

The initialization part computes the slicing seed for the fixpoint by independently applying the intraprocedural slicing $\$(\_,\_,\_)$ with the slicing criterion sc for each function in the program $\mathfrak{p}$. The notation $A {\cup}{=} B$ (line 3) stands for "A becomes $A \cup B$" where $\cup$ is the set union. Similarly, $A \uplus= B$ (line 4)

```
termSlicing
Input: $sc, \mathfrak{F}_\mathfrak{p}, SE, CU$
Output: $slicedFnSet, dfsc$
$workingSet' := \emptyset$; $dfsc := \emptyset$;
for all $fn(args)\{fnBody\} \in \mathfrak{F}_\mathfrak{p}$ do
   $SCinit :: fn\langle fnInitSlice\rangle := \$(fnBody, \{x \in sc \mid x \preccurlyeq fs$ or $x \preccurlyeq args\}, SE)$;
   $workingSet' \cup= \{SCinit :: fn\langle fnInitSlice\rangle\}$;
   $dfsc \uplus= SCinit$;
od
$workingSet := \emptyset$;
while $workingSet \neq workingSet'$ do
   $workingSet := workingSet'$;
   for all $SC :: fn\langle fnSlice\rangle \in workingSet$ do
      $wsFnCalled := \emptyset$;
      for all $Call \in CU$ for all $Call\ fnCalled \preccurlyeq fnSlice$ do
         $fnCldSC := SC\ {}^{fn}\lfloor_{fnCalled}$;
         for all $fnCldSCPrev :: fnCalled\langle\_\rangle \in workingSet$ do
            if $fnCldSC \sqsubseteq fnCldSCPrev$ then break;
         $fnCldBd :=$ getFnBody$(fnCalled, \mathfrak{F}_\mathfrak{p})$;
         $fnCldSCNew :: fnCalled\langle fnCldSlice\rangle := \$(fnCldBd, fnCldSC, SE)$;
         $wsFnCalled \cup= \{fnCldSCNew :: fnCalled\langle fnCldSlice\rangle\}$;
         $SC \uplus= fnCldSCNew\ {}_{fnCalled}\rfloor^{fn}$;
      od
      $wsFnCalling := \emptyset$;
      for all $Call \in CU$ for all $fnCalling \in \mathfrak{F}_\mathfrak{p}$ s.t. $Call\ fn \preccurlyeq fnCalling$ do
         $fnClgSC := SC\ {}_{fn}\lceil^{fnCalling}$;
         for all $fnCallingSCPrev :: fnCalling\langle\_\rangle \in workingSet$ do
            if $fnClgSC \sqsupseteq fnCallingSCPrev$ then break;
         $fnClgBd :=$ getFnBody$(fnCalling, \mathfrak{F}_\mathfrak{p})$;
         $fnClgSCNew :: fnCalling\langle fnClgSlice\rangle := \$(fnClgBd, fnClgSC, SE)$;
         $wsFnCalling \cup= \{fnClgSCNew :: fnCalling\langle fnClgSlice\rangle\}$;
         $SC \uplus= fnClgSCNew\ {}^{fnCalling}\rceil_{fn}$;
      od
      $fnBd :=$ getFnBody$(fn, \mathfrak{F}_\mathfrak{p})$;
      $SCNew :: fn\langle fnSliceNew\rangle := \$(fnBd, SC, SE)$;
      $dfsc \uplus= SCNew$;
      for all $Call \in CU$ for all $Call\ fnCalled \preccurlyeq fnSliceNew$ do
         if $\_ :: fnCalled\langle\_\{\}\rangle \in wSetFnCalled$ then
            $fnSliceNew :=$ erraseSubterm$(Call\ fnCalled, fnSliceNew)$
      od
      $workingSet' \Cup= \{SCNew :: fn\langle fnSliceNew\rangle\} \Cup wsFnCalled \Cup wsFnCalling$;
   od
od
$slicedFnSet :=$ get$\langle\rangle$Content$(workingSet)$
```

Fig. 3. Program slicing as term slicing algorithm

is the union of two data dependency graphs. Namely, $A \uplus B$ is the set union for graph edges filtered by the criterion that if a variable v is independent in A but dependent in B (i.e. there exists an edge $\_\curvearrowleft\_$ with v on one of the ends) then the independent variable v is eliminated from $A \uplus B$. For example, the initialization step applied on the program in Fig. 1 produces the following *workingSet′*: $\texttt{z} :: \texttt{Inc}\langle\texttt{Inc(z)}\{\texttt{Call Add(z,}\sharp\texttt{)}\}\rangle, \emptyset :: \texttt{Main}\langle\texttt{Main()}\{\}\rangle, \emptyset :: \texttt{A}\langle\texttt{A(}\sharp,\sharp\texttt{)}\{\}\rangle, \emptyset :: \texttt{Add}\langle\texttt{Add(}\sharp,\sharp\texttt{)}\{\}\rangle$.

The fixpoint loop (lines 7-39) discovers the call graph in an on-demand fashion using the context-update set CU, which directs the fixpoint iteration towards applying the slicing on the called/calling function. As such, when a context-update (e.g. `Call_(_)` in the semantics of WhileF) is encountered in the current slice, we proceed to slice *the called function* (lines 10-19). Next, when a context-update of the currently considered functions is encountered, we proceed again to slice *the calling function* (lines 20-29). Each time we update the current data-flow-augmented slicing criterion and the slice of the current function (lines 30-36). For example, the discovery of the call graph starting with the function $\texttt{Inc(z)}\{\texttt{Call Add(z,}\sharp\texttt{)}\}$ in the program from Fig. 1 adds, during the first iteration of the fixpoint-loop, the called function $\texttt{Add(a,b)}\{\texttt{a} := \texttt{a} + \texttt{b}\}$ and the calling function $\texttt{A(}\sharp\texttt{,y)}\{\texttt{Call Inc(y)}\}$. We iterate this process until the skeleton subterm of every function is reached, i.e., *workingSet* is stable, e.g., see the result from Fig. 2. Note that the stability of *workingSet* induces the stability of *dfsc*, the data flow augmented slicing criterion.

We now describe in more details each of the three parts of the fixpoint loop: the *called* (lines 10-19), the *calling* (lines 20-29), and the *current* (lines 30-36) functions. The *called* and *calling* parts have a similar flow with slight differences in the operators used. They can be summarized as:

$$SC \uplus\!= SC\ {}^{fn}\lfloor_{fnCalled}\ \mathit{filtered}\$(\mathit{fnCalled}, \sqsubseteq)\ {}_{fnCalled}\rfloor^{fn}$$

$$SC \uplus\!= SC\ {}_{fn}\lceil^{fnCalling}\ \mathit{filtered}\$(\mathit{fnCalling}, \sqsupseteq)\ {}^{fnCalling}\rceil_{fn}$$

where *fn* is the name of the current function, *fnCalled* is the name of a functions called from *fn*, and *fnCalling* is the name of a function which is calling *fn*.

The operators ${}^{-}\lfloor_{-}$ and ${}_{-}\rfloor^{-}$ stand for the abstraction of the slicing criterion *downwards in the calling graph* from *fn* into *fnCalled* and back, respectively. The abstraction ${}^{fn}\lfloor_{fnCalled}$ pivots on the actual parameters of *fnCalled* and, based on patterns of function calls, it maps the actual parameters of *fnCalled* from the current environment $SC :: fn$ into the environment of *fnCalled*. The abstraction ${}_{fnCalled}\rfloor^{fn}$ renders the reverse mapping from the (sliced) called environment back into the current one. Similarly for the ${}_{-}\lceil^{-}$ and ${}^{-}\rceil_{-}$ operators, which perform the abstraction *upwards in the call graph* from *fn* to *fnCalling*, pivoting on the parameters of *fn*. For example, for program `Px` from Fig. 1 we have $\overset{\curvearrowleft}{\texttt{z},\texttt{i}}\ {}^{\texttt{Inc}}\lfloor_{\texttt{Add}}\ \overset{\curvearrowleft}{\texttt{a},\texttt{b}}\ {}_{\texttt{Add}}\rfloor^{\texttt{Inc}}\ \overset{\curvearrowleft}{\texttt{z},\texttt{i}}$ and $\overset{\curvearrowleft}{\texttt{z},\texttt{i}}\ {}_{\texttt{Inc}}\lceil^{\texttt{A}}\ \overset{\curvearrowleft}{\texttt{y},\sharp} = \texttt{y}\ {}^{\texttt{A}}\rceil^{\texttt{Inc}}\ \overset{\curvearrowleft}{\texttt{z},\texttt{i}}$. For the current work, the only pattern of function calls that we have experimented is the *complete list of call-by-reference parameters*.

The operator *filtered*(fnC, rel) (lines 13-17 and 23-27) is a *filtered* slicing of fnC, where the filter is a relation between the current abstraction of SC and previously computed slicing criterions for the called/calling function fnC. We say that $SC \sqsubseteq SCPrev$ if SC is a subgraph of $SCPrev$ such that there is no edge

$\overset{\curvearrowleft}{v, v'}$ in *SCPrev* where v is a node in SC and v' is a function parameter which is not in SC. This means that SC has no additional dependent data v' among the function parameters that should participate to the current slicing criterion. For example, this relation is exploited for the call of function `Add` in function `A` from Fig. 1. Namely, the algorithm discovers the relation $\overset{\curvearrowleft}{\mathtt{a}, \mathtt{b}}$ for `Add`'s parameters upon the call of `Add` from function `Inc`. Later, when function `Add` is called in `A` only with parameter b, the subgraph relation $\mathtt{b} \sqsubseteq \overset{\curvearrowleft}{\mathtt{a}, \mathtt{b}}$ shows that the already sliced $\mathtt{Add}(\mathtt{a}, \mathtt{b})$ contains the slice of $\mathtt{Add}(\sharp, \mathtt{b})$ hence, there is nothing else to be done for this function. Meanwhile, $SC \sqsupseteq SCPrev$ is defined as $SCPrev \sqsubseteq SC$ due to the fact that now the sense in the dependency graph is reversed and so the slicing criterion in the calling function (*SCPrev*) is the one to drive the reasoning. Hence, if the filter relation is true then the new slice is not computed anymore (lines 14 and 24) because the current slicing criterion is subsumed by the previous computation.

In lines 30-36 we compute a new slice for the current function *fn* and in line 37 we collect the slices currently computed for the program functions.

Lines 30-36 are more of a beautification of the slice of the currently sliced function *fn*. This beautification is made by the elimination from the slice of any context-update subterm *Call fnCalled* having an empty body for the currently computed slice (lines 33-36). For example, the call to function `Add` from function `A`, i.e., $\mathtt{CallAdd}(\sharp, \mathtt{b})$, is eliminated from the slice computed for function `A` due to the emptiness of the sliced body of function `Add` starting with the slicing criterion b. Note that this fact can be concluded only from the data-flow relation among the parameters of a function, provided that we add a special symbol ℓ for the local variables such that any function parameter v depending on some local variable is going to appear as connected to ℓ, i.e., either $\overset{\curvearrowleft}{v, \ell}$ or $\overset{\curvearrowleft}{\ell, v}$. Namely, in function `Inc` from Fig. 1 we have $\overset{\curvearrowleft}{\mathtt{z}, \ell}$ due to the fact that `Add` brings $\overset{\curvearrowleft}{\mathtt{z}, \mathtt{i}}$ in the SC of `Inc`. However, in what follows we do not insist on the data dependency on local variables in order not to burden the notation.

Finally, in line 37 we collect all the slices computed at the current iteration in *workingSet*$'$. Note that $\_\Cup\_$ operator from line 37 is an *abstract union* which first computes the equivalence class of slices for each function, based on the graph inclusion of the data-flow-augmented slicing criterion, and then performs the union of the results. Namely, if there is a function with F with three parameters x, y, z such that $\overset{\curvearrowleft}{x, y}$ and z is independent, and if at some iteration of the fixpoint we have the slice $F(x, y, \sharp)\{B_{x,y,\sharp}\}$ in the set *wsFnCalled* and $F(\sharp, \sharp, z)\{B_{\sharp,\sharp,z}\}$ in the set *wsFnCalling*, then in *workingSet*$'$ we have $Fx, y, z\{B_{x,y,z}\}$ where in $B_{x,y,z}$ we put together the two skeletons $B_{x,y,\sharp}$ and $B_{\sharp,\sharp,z}$.

Recall that, in Section 3, we described two interprocedural slicing methods presented in [26] and [11], being the second one more precise than the first one. In our approach the difference is based solely on the data flow relation we use for \$. Hence, we can distinguish two types of **termSlicing**: the *naïve* one where the data flow relations are ignored and the *savvy* one which collects and uses data flow relations. Note that the data flow relation is currently assumed as given.

```
Main () {
 sum := 0;
 Local i, j;
 i := 1; j := −1;
 While i < 11 Do
  Call A (sum, i);
  Call B (sum, j);
  Call A (j, i)
}

A (x, y) {
 If x > 1 Then
  Call Add(x, y);
  Call Inc (y)
}
B (x, y) {
 If x > 0 Then
  Call B(x + y, y)
}

Add (a, b) {
 a := a + b
}
Inc (z) {
 Local i, j;
 i := 1; j := i;
 Call Add (z, i);
 Call Inc (j)
}
```

Fig. 4. PX—the extension of the WhileF program Px

For example, the iterations of the savvy **termSlicing** for the program PX in Fig. 4 and the slicing criterion {z} are listed in Fig. 5. Namely, in the first boxed rows the slicing criterion {z} is applied on $\mathfrak{F}_{\mathtt{PX}}$ to produce the skeleton subterms used as the fixpoint seed. Hence, the fixpoint seed contains one nonempty skeleton as z appears only in Inc. Note that i—the second parameter of Call Add—is abstracted to $\sharp$ as no data dependency is currently determined for it.

In the second box of rows we consider the slicing criterion for Inc—the only one nonempty from the seed—and we iterate the fixpoint for it. The first row deals with the (only) called function appearing in Inc's skeleton, namely $\mathtt{Add}(\mathtt{z}, \sharp)$. Note that the slicing criterion z is abstracted downwards in the call graph so the slicing criterion becomes a, the first formal parameter of Add. The slice of Add with {a} as slicing criterion is showed in the third column while the slicing criterion becomes $\overset{\curvearrowleft}{\mathtt{a},\mathtt{b}}$, i.e., a depends on b. Because b is a formal parameter, it gets abstracted back in Inc as Add's actual parameter i. Hence, the updated criterion used in Inc is $\overset{\curvearrowleft}{\mathtt{z},\mathtt{i}}$ and it is used for the calling function A, in the second row, and also for the recursive call to Inc itself, in the third row. In these rows, the slicing criterion is abstracted upwards in the call graph and the formal parameter z becomes y in A and j in Inc. Meanwhile i is ruled out (becomes $\sharp$) because it is not a parameter and hence it is not relevant in a calling function. The fourth row shows the computation of Inc's skeleton based on the current slicing criterion $\overset{\curvearrowleft}{\mathtt{z},\mathtt{i}}$. Furthermore, upon performing the abstract union $\Cup$ at the end of the fixpoint iteration, then Inc's skeleton is:

```
Inc(z){Local i, j; i := 1; j := i; Call Add (z, i); Call Inc (j)}
```

The fixpoint iteration continues in the third box by adding to the slice the function Main due to the upward phase (since Main contains a call to A). The upward parameter substitution of y from A is i in Main and the slice of Main is updated in the third row. Note that the $\Box$ in all the other rows signifies the reach of the **break** in lines 14 or 24 in **termSlicing** and stands for "nothing to be done." The fourth box contains the final step of the fixpoint when there is nothing else changed in *workingSet′* (i.e. all the rows contain $\Box$ in the last column). Hence, for the example in Fig. 4 we obtain the slice in Fig. 2 with the only difference that the sliced Inc is now the entire Inc from Fig. 4 (due to the newly added assignment "j:=i").

Slicing variables	**Function contexts**	**Computed slice** (identified subterms)
$\mathtt{z} :: \top$	${}^{\top}\lfloor_{\mathtt{Inc}}\ \mathtt{z} \to \mathtt{z}\ {}_{\mathtt{Inc}}\rfloor^{\top}$	$\mathtt{Inc(z)}\ \{\mathtt{Call\ Add(z,\sharp)}\}$
	${}^{\top}\lfloor_{\mathtt{Main}}\ \sharp = \emptyset\ {}_{\mathtt{Main}}\rfloor^{\top}, \ldots$	$\mathtt{Main()}\{\}, \mathtt{A(\sharp,\sharp)}\{\}, \mathtt{B(\sharp,\sharp)}\{\}, \mathtt{Add(\sharp,\sharp)}\{\}$
$\mathtt{z} :: \mathtt{Inc}$	${}^{\mathtt{Inc}}\lfloor_{\mathtt{Add}}\ \mathtt{a} \to \overset{\frown}{\mathtt{a,b}}\ {}_{\mathtt{Add}}\rfloor^{\mathtt{Inc}}\ \overset{\frown}{\mathtt{z,i}}$	$\mathtt{Add(a,b)}\ \{\mathtt{a := a + b}\}$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{A}}\ \overset{\frown}{\mathtt{y},\sharp} \to \overset{\frown}{\mathtt{y},\sharp}\ {}^{\mathtt{A}}\rceil_{\mathtt{Inc}}\ \overset{\frown}{\mathtt{z,i}}$	$\mathtt{A(\sharp,y)}\ \{\mathtt{Call\ Inc(y)}\}$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{Inc}}\ \overset{\frown}{\mathtt{j},\sharp} \to \overset{\frown}{\mathtt{j,i}}\ {}^{\mathtt{Inc}}\rceil_{\mathtt{Inc}}\ \overset{\frown}{\mathtt{z,i}}$	$\mathtt{Inc(\sharp)}\{\mathtt{Local\ i,j; i{:=}1; j{:=}i; Call\ Add(\sharp,i); Call\ Inc(j)}\}$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	$\overset{\frown}{\mathtt{z,i}} \to \overset{\frown}{\mathtt{z,i}}$	$\mathtt{Inc(z)}\{\mathtt{Local\ i;\ i{:=}1;\ Call\ Add(z,i)}\}$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Add}}\ \mathtt{b} \sqsubseteq \overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}\ {}_{\mathtt{Add}}\rfloor^{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Inc}}(\mathtt{z} \sqsubseteq \overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc})\ {}_{\mathtt{Inc}}\rfloor^{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}_{\mathtt{A}}\lceil^{\mathtt{Main}}\ \mathtt{i} \to \mathtt{i}\ {}^{\mathtt{Main}}\rceil_{\mathtt{A}}\ \mathtt{y}$	$\mathtt{Main()}\{\mathtt{Local\ i;i{:=}1;While\ i{<}1\ Do\ Call\ A(\sharp,i)}\}$
$\mathtt{y} :: \mathtt{A}$	$\mathtt{y} = \mathtt{y} :: \mathtt{A}$	$\square$
$\overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{Inc}}(\overset{\frown}{\mathtt{z,i}} \sqsupseteq \overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc})\ {}^{\mathtt{Inc}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{A}}(\overset{\frown}{\mathtt{x,y}} \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	$\overset{\frown}{\mathtt{a,b}} = \overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	$\square$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	${}^{\mathtt{Inc}}\lfloor_{\mathtt{Add}}(\overset{\frown}{\mathtt{a,b}} \sqsubseteq \overset{\frown}{\mathtt{a,b}} :: \mathtt{Add})\ {}_{\mathtt{Add}}\rfloor^{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{A}}(\overset{\frown}{\mathtt{y},\sharp} \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	$\overset{\frown}{\mathtt{z,i}} = \overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Add}}(\mathtt{b} \sqsubseteq \overset{\frown}{\mathtt{a,b}} :: \mathtt{Add})\ {}_{\mathtt{Add}}\rfloor_{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Inc}}(\mathtt{z} \sqsubseteq \overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc})\ {}_{\mathtt{Inc}}\rfloor^{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}_{\mathtt{A}}\lceil^{\mathtt{Main}}(\mathtt{i} \sqsupseteq \mathtt{i} :: \mathtt{Main})^{\mathtt{Main}}\rceil_{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	$\mathtt{y} = \mathtt{y} :: \mathtt{A}$	$\square$
$\overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{Inc}}(\overset{\frown}{\mathtt{z,i}} \sqsupseteq \overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc})\ {}^{\mathtt{Inc}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{A}}(\overset{\frown}{\mathtt{x,y}} \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	$\overset{\frown}{\mathtt{a,b}} = \overset{\frown}{\mathtt{a,b}} :: \mathtt{Add}$	$\square$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	${}^{\mathtt{Inc}}\lfloor_{\mathtt{Add}}(\overset{\frown}{\mathtt{a,b}} \sqsubseteq \overset{\frown}{\mathtt{a,b}} :: \mathtt{Add})\ {}_{\mathtt{Add}}\rfloor^{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{A}}(\overset{\frown}{\mathtt{y},\sharp} \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	$\overset{\frown}{\mathtt{z,i}} = \overset{\frown}{\mathtt{z,i}} :: \mathtt{Inc}$	$\square$
$\mathtt{i} :: \mathtt{Main}$	${}^{\mathtt{Main}}\lfloor_{\mathtt{A}}(\mathtt{y} \sqsubseteq \mathtt{y} :: \mathtt{A})\ {}_{\mathtt{A}}\rfloor^{\mathtt{Main}}\ \mathtt{i}$	$\square$
$\mathtt{i} :: \mathtt{Main}$	${}^{\mathtt{Main}}\lfloor_{\mathtt{B}}(\sharp \sqsubseteq \emptyset :: \mathtt{B})\ {}_{\mathtt{B}}\rfloor^{\mathtt{Main}}\ \mathtt{i}$	$\square$
$\mathtt{i} :: \mathtt{Main}$	$\mathtt{i} = \mathtt{i} :: \mathtt{Main}$	$\square$

Fig. 5. Program slicing as term slicing - the fixpoint iterations

termSlicing terminates because there exists a finite set of function skeleton subterms, a finite set of data flow graphs, a finite set of edges in the call graph for each function, and any loop in the call graph is solved based on the data flow graph ordering. Moreover, **termSlicing** produces a valid slice because it exhaustively saturates the slicing criterion. However, the obtained slice is not minimal due to the skeletons union $\Cup$. Still, there is a consistent difference between the *naïve* and the *savvy* methods. In order to achieve a better degree of minimality we have to apply abstractions on the data-flow-augmented slicing criterion.

4.2 System Description

We briefly present in this section our prototype which is implemented in Maude [3]. The source code is available at `http://maude.sip.ucm.es/slicing/`. A key distinguishing feature of Maude is its systematic and efficient use of reflection (i.e. Maude's capability of handling and reasoning about terms that represent specifications described in Maude itself) through its predefined `META-LEVEL` module [3, Chapter 14]. We have used these features to implement a tool that receives a set of definitions, a sort where the computations take place, and a set of slicing variables. Since all these elements can be used as usual data, we can traverse the semantic rules, analyze them, and execute the program using them. Note that the user has to provide the rules responsible for context-update while the parameter passing operators $\_\lfloor\_\ \_\rfloor\_$ and $\_\lceil\_\ \_\rceil\_$ are particularized here to an all-parameters-ordered-pass-by-reference pattern.

The tool is started by loading into Maude the `slicing.maude` file available at the webpage above. It starts an input/output loop where modules and commands can be introduced by enclosing them in parentheses. Once the module with the semantics has been loaded, we have to introduce `ESt`, the sort for the mapping between variables and values, and `RWBUF`, the sort for the read/write buffer, as the sorts responsible for the side effects. Similarly, we indicate that `CallF` is the rule for context-update:

```
 Maude> (set side-effect sorts ESt RWBUF .)
 ESt RWBUF selected as side effect sorts.
 Maude> (set context-update rules CallF .)
 CallF selected as context-update rules.
```

We can now start the slicing process by indicating that `Statement` is the sort for instructions, `myFuns` is a constant standing for the definition of the functions `Main`, `A`, `Add`, and `Inc` from Figure 4, and `z` is the slicing variable. The tool displays the relevant variables and the sliced code for each function as:

```
Maude> (islice Statement with defs myFuns wrt z .)
The variables to slice 'Inc are {i, j, z}
'Inc(z){
   Local i ; Local j ;
   i := _ ; j := _ ;
   Call 'Add(z,i);
   Call 'Inc(j)
}
...
```

We test our proposed method for interprocedural slicing on a set of benchmarks addressing embedded and real-time applications. As such, we use a set of small examples, grouped under the name `bundle`, from a survey [24] on program slicing techniques, automatically-generated code from typical Scade designs [5], as well as a standard set of real-time benchmarks—called PapaBench [15]. In Figure 6, each program is identified by name, a short description, size parameters (LOC, number of functions, #funs, and function calls, #calls), and the

Name	**Program Description**	**LOC**	**#funs**	**#calls**	**red (%)**
`bundle`	A collection of (extended) examples from [24]	71	7	25	38 %
`selector_2`	Generated code from SCADE design - 2 SSM	426	6	11	91 %
`selector_3`	Generated code from SCADE design - 3 SSM	455	7	19	85 %
`autopilot`	PapaBench - autopilot	1384	95	214	74 %
`fbw`	PapaBench - fly_by_wire	638	41	110	78 %

Fig. 6. Set of benchmark programs for interprocedural slicing

average reduction in the number of statements, for several runs with different sets of slicing variables. This reduction shows that the methodology works better on bigger programs (the `bundle`, with very small examples, presents the lowest reduction, because all variables are closely related). The Scade benchmarks, explained below, present the greatest reduction because the variables have very specific behaviors, hence allowing a very efficient use of slicing.

The Scade Suite development platform [5] is a mixed synchronous language, combining variants of Lustre [9] (i.e. data-flow) and Esterel [2] (i.e. control-flow). Scade facilitates the design of embedded and real-time systems in a modular fashion, and the modularity is preserved in the generated C code. The two Scade designs—`selector_2` and `selector_3`—consist of two, and respectively three, parallel state machines (called SSM - Safe State Machines) which embed in their states calls to external functions and constrain (via shared variables) how these state machines communicate among them.

PapaBench is extracted from an actual real-time system for Unmanned Aerial Vehicle (UAV) and consists of two programs `fly_by_wire` and `autopilot`, designed to run on different processors. The application consists of a number of tasks which are executed in a control loop. For example, the `autopilot` program focuses on the UAV airframe and has eight different tasks (e.g. for controlling the navigation, stabilisation, altitude or communication - radio or GPS).

We test our interprocedural slicing at the level of the entire program as well as at the level of each task. Let us consider the function `radio_control_task` (in `autopilot`) which manages radio orders based on various operation modes (e.g. PITCH, ROLL, THROTTLE, etc) and sets new values for several flight parameters (e.g. *desired_roll* or *desired_pitch*). This particular function has a call graph of about 21 nodes. We could use, for example, a slicing criterion which consists of all program variables used in `radio_control_task` in order to investigate the tasks which are depending (i.e. their intraprocedural slice is not empty) or not on the computation of `radio_control_task`. The interprocedural slice shows a dependence of the `radio_control_task` with tasks such as `altitude_control_task` and `climb_control_task`, which rely on global flight parameters used by the radio controller. This testing strategy is applied on all benchmarks and, together with the resulting traces and the Scade designs, are available on the tool webpage at `http://maude.sip.ucm.es/slicing/`.

5 Concluding Remarks and Ongoing Work

The formal language definitions based on the rewriting logic framework support program executability and create the premises for further development of program analyzers. In this paper we have presented a generic algorithm for interprocedural slicing based on results of meta-level analysis of the language semantics. In summary, the slicing prerequisites are: side-effect and context-update language constructs with data flow information for the side-effect constructs and parameter passing patterns for the context-update constructs. The actual program slicing computation, presented in the current work, is done through term slicing and is meant to set the aforementioned set of prerequisites. This work complements the recent advances in semantics-constructed tools for debugging [18], automated testing [16], and program analysis [17].

From the prototype point of view, we also plan to investigate the automatic inference of the newly identified slicing prerequisites, i.e., meta-analysis for context-updates deduction and parameter passing pattern inference. This would greatly simplify the user task, since he will just introduce the program and the slicing criterion and the tool would be in charge of computing all the required constructors. We also have to further develop the already existing side-effect extraction with data flow information. Finally, we aim to develop the method for language semantics defined in Maude but also in $\mathbb{K}$ [19].

References

1. 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, vol. 6803, pp. 34–48. Springer, Heidelberg (2011)
2. Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program (SCP) 19(2), 87–152 (1992)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Ellison, C., Rosu, G.: An executable formal semantics of c with applications. In: POPL, pp. 533–544 (2012)
5. Esterel Technologies Scade Language Reference Manual 2011 (2011)
6. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of java programs in javaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
7. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)

8. Harman, M., Danicic, S.: Using program slicing to simplify testing. Journal of Software Testing, Verification and Reliability 5, 143–162 (1995)
9. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. In: Proc. of the IEEE, pp. 1305–1320 (1991)
10. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. John Wiley & Sons (1990)
11. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Conference on Programming Language Design and Implementation, PLDI 1988, pp. 35–46 (1988)
12. 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)
13. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theor. Comput. Sci. 285(2), 121–154 (2002)
14. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
15. Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.P., De Michiel, M.: PapaBench: a Free Real-Time Benchmark. In: WCET 2006 (2006)
16. 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)
17. Riesco, A., Asăvoae, I.M., Asăvoae, M.: A generic program slicing technique based on language definitions. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 248–264. Springer, Heidelberg (2013)
18. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Journal of Logic and Algebraic Programming (2012)
19. Şerbănuţă, T., Ştefănescu, G., Roşu, G.: Defining and executing P systems with structured data in K. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 374–393. Springer, Heidelberg (2009)
20. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis, pp. 189–233 (1981)
21. Silva, J., Chitil, O.: Combining algorithmic debugging and program slicing. In: PPDP, pp. 157–166. ACM Press (2006)
22. Sridharan, M., Fink, S.J., Bodík, R.: Thin slicing. In: PLDI, pp. 112–122 (2007)
23. 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)
24. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)
25. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67, 226–293 (2006)
26. Weiser, M.: Program slicing. In: Proc. of the 5th International Conference on Software Engineering, ICSE 1981, pp. 439–449. IEEE Press (1981)