Document downloaded from:

http://hdl.handle.net/10251/47187

This paper must be cited as:

Alpuente Frasnedo, M.; Ballis, D.; Frechina, F.; Romero, DO. (2014). Using conditional trace slicing for improving Maude programs. Science of Computer Programming. 80:385-415. doi:10.1016/j.scico.2013.09.018.

The final publication is available at

http://dx.doi.org/10.1016/j.scico.2013.09.018

Copyright Elsevier

main Manuscript
Click here to view linked References

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Using Conditional Trace Slicing for improving Maude programs☆

María Alpuente[a], Demis Ballis[b], Francisco Frechina[a], Daniel Romero[a]

[a] *DSIC-ELP, Universitat Politècnica de València, Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain.*
[b] *Dipartimento di Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy.*

Abstract

Understanding the behavior of software is important for the existing software to be improved. In this paper, we present a trace slicing technique that is suitable for analyzing complex, textually-large computations in rewriting logic, which is a general framework efficiently implemented in the Maude language that seamlessly unifies a wide variety of logics and models of concurrency. Given a Maude execution trace $\mathcal{T}$ and a slicing criterion for the trace (i.e., a piece of information that we want to observe in the final computation state), we traverse $\mathcal{T}$ from back to front and the backward dependence of the observed information is incrementally computed at each execution step. At the end of the traversal, a simplified trace slice is obtained by filtering out all the irrelevant data that were found not to influence the data of interest. By narrowing the size of the trace, the slicing technique favors better inspection and debugging activities since most tedious and irrelevant inspections that are routinely performed during diagnosis and bug localization can be eliminated automatically. Moreover, cutting down the execution trace can expose opportunities for further improvement, which we illustrate by means of several examples.

Keywords: trace slicing, program debugging and comprehension, model checking, rewriting logic

1. Introduction

The analysis of computation traces plays an important role in many program analysis approaches. Software systems commonly generate large and complex execution traces, whose analysis (or even simple inspection) is extremely time-consuming and, in

☆This work has been partially supported by the EU (FEDER) and the Spanish MEC TIN2010-21062-C02-02 project, by Generalitat Valenciana, ref. PROMETEO2011/052. Also, D. Romero is supported by FPI-MEC grant BES-2008-004860 and F. Frechina is supported by FPU-ME grant AP2010-5681.

Email addresses: `alpuente@dsic.upv.es` (María Alpuente), `demis@dimi.uniud.it` (Demis Ballis), `ffrechina@dsic.upv.es` (Francisco Frechina), `dromero@dsic.upv.es` (Daniel Romero)

some cases, unfeasible to perform by hand. Trace slicing is a technique for reducing the size of execution traces by focusing on selected execution aspects, which makes it suitable for trace analysis and monitoring [1] and is also helpful for program debugging, improvement, and understanding [2, 3].

Rewriting Logic (RWL) is a logic of change that is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [4] and Web systems [5, 6]). RWL is efficiently implemented in the high-speed rewriting language Maude [7]. Roughly speaking, a Maude program consists of a *(conditional) term rewriting system* (CTRS), together with an *equational theory* (also possibly conditional) that may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are applied *modulo* the equations and algebraic axioms. Rewriting logic-based tools, like the Maude-NPA protocol analyzer, the Maude LTLR model checker, and the Java PathExplorer runtime verification tool (just to mention a few [8]), are used in the analysis and verification of programs and protocols wherein the system states are represented as algebraic entities (elements of an algebraic data type that is specified by the equational theory) while the system computations are modelled via the rewrite rules, which describe transitions between states and are performed *modulo* the equations and axioms. The execution traces produced by these tools are usually very complex and are therefore not amenable to manual inspection. However, not all the information that is in the trace is needed to analyze a given piece of information in a target state. For instance, consider the following rules that define (a part of) the standard semantics of a simple imperative language: 1) `crl <while B do I, St> => <skip, St> if <B, St> => false /\ isCommand(I)`, 2) `rl <skip, St> => St`, and 3) `rl <false, St> => false`. Then, in the two-step execution trace `<while false do X := X + 1, {}>` → `<skip, {}>` → {}, we can observe that the statement `X := X + 1` is not relevant to compute the output {}. Therefore, the trace could be simplified by replacing `X := X + 1` with a special variable • and by enforcing the logical compatibility condition `isCommand(•)`. This way we guarantee the correctness of the simplified trace. In other words, any concretization of the simplified trace (which instantiates the variable • and meets the compatibility condition) is a valid trace that still generates the target data that we are observing (in this case, the output {}).

The basic debugging aid provided by Maude consists of a forward tracing facility that allows the user advance through the program execution, letting the user select the statements being traced. By manually controlling the traceable equations or rules, the displayed trace view can be reduced. However, the user can easily miss the general overview because all rewrite steps that are obtained by applying the equations or rules for the selected function symbols (including all evaluation steps for the conditions of such equations/rules) are shown in the output execution trace, whereas the algebraic axiom applications are not recorded in the trace. Thus, the trace is both, huge and incomplete, and when the user detects an erroneous intermediate result, it is difficult to determine where the incorrect inference started. Moreover, this trace is either directly displayed or written to a file (in both cases in plain text format) thus being only amenable for user

inspection. This is in contrast with the trace slices described below, which are small, automatically generated, and complete (all steps are recorded by default). Moreover, they can be directly displayed or delivered in their meta-level representation and only consist of the information that is strictly needed to deliver the critical part of observed results.

The backward tracing approach developed in this paper aims to improve program analysis and debugging in rewriting-based programming by helping the user to *think backwards* –i.e., to deduce the conditions under which a program produces the incorrect output. In conventional programming environments, for such an analysis the programmer must repeatedly determine which statements in the code have an influence on the value of a given parameter at a given call, which is usually done manually without any assistance from the debugger. By developing an appropriate notion of antecedents for RWL, our trace slicing technique tracks back reverse dependences and causality along execution traces and then cuts off irrelevant information that does not influence the data observed from the trace. In other words, when execution is stopped at a given computation point (typically, from the location where a fault is manifested), we are able to undo the effect of the last statement executed on the selected data by issuing the step-back facility provided by our slicer (for a given slicing criterion). Thus, by stepwisely reducing the amount of information to be inspected, it is easier for the user to locate errors because many computation steps (and the corresponding program statements involved in the step) can be ignored in the process of localizing the program fault area. Moreover, during the trace slice computation, different types of information are computed that are related to the program execution, for example, contributing actions and data and noncontributing ones. After computation of a trace slice, all the noncontributing information is discarded from the trace, and we can even take advantage of the filtered information for the purpose of dynamic program slicing.

Contributions. This article offers an up-to-date, comprehensive, and uniform presentation with examples of the backward tracing slicing methodology as developed in [2, 9, 10, 11].

Our proposal for conditional trace slicing is aimed at endowing the RWL framework with a new instrument that can be used to improve the Maude programs, including any RWL-based tool that produces or manipulates RWL computations (e.g., Maude execution traces). The contributions of the paper can be summarized as follows:

1. We describe the first slicing technique for Maude programs that can be used to drastically reduce complex, textually-large system computations w.r.t. a user-defined slicing criterion that selects those data that we want to track back from a given point. The distinguishing features of our technique are as follows:
    - (a) The system copes with the most prominent RWL features, including algebraic axioms such as associativity, unity, and commutativity.
    - (b) The system also copes with the extremely rich variety of conditions that occur in Maude theories (i.e., equational conditions $s = t$, matching conditions

$p := t$, and rewrite expressions $t \Rightarrow p$) by taking into account the precise way in which Maude mechanizes the conditional rewriting process so that all those rewrite steps are revisited backwards in an instrumented, fine-grained way.

(c) Unlike previous backward tracing approaches [9, 12], which are based on a costly, dynamic labeling procedure, here we use a less expensive, incremental technique of matching refinement that allows the relevant data to be traced back.

2. We have developed a tool, called Julienne that implements our conditional slicing technique, and we have carried out a number of experiments that confirm the usefulness of our approach. Julienne is the first slicing tool that can be used to analyze execution traces of RWL-based programs and tools. Julienne greatly reduces the size of the execution traces (up to 98% of the original size according to our experiments), thus making their analysis feasible even in the case of complex, real-size problems. The implementation comprises a front-end consisting of a Web graphical user interface and a back-end consisting of a Maude implementation that uses Maude meta-level capabilities. The tool is publicly available at `http://users.dsic.upv.es/grupos/elp/soft.html`.
3. To give the reader a better feeling of the generality and wide application range of our conditional slicing approach, we describe some applications to debugging and improvement of Web systems and protocol specifications.

Plan of the paper. The rest of the paper is organized as follows. Section 2 recalls some fundamental notions of RWL, and Section 3 summarizes the conditional rewriting modulo equational theories defined in Maude. In Section 4, the backward conditional slicing technique is formalized by means of a transition system that traverses the execution traces from back to front. Section 5 describes Julienne, which is the trace slicer that implements our theoretical framework, together with some experiments that have been performed with our tool. We also discuss the use of trace slicing to improve program analysis, debugging and comprehension in several application domains. The related work is discussed in Section 7, and Section 8 concludes.

2. Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [12] and Rewriting Logic [13]. Some familiarity with the Maude language [7] is also required. Maude [14] is a rewriting logic [13] specification and verification system whose operational engine is mainly based on a very efficient implementation of rewriting. A Maude program consists of a composition of modules containing sort and operator declarations, as well as equations relating terms over the operators and universally quantified variables. Maude notation will be introduced "on the fly" as needed in examples.

2.1. The term-language of Maude

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [7]. We assume an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. $\tau(\Sigma, \mathcal{V})_s$ and $\tau(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. We write $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. In order to simplify the presentation, we often disregard sorts when no confusion can arise.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Positions are ordered by the prefix ordering, that is, given the positions w_1 and w_2, $w_1 \leq w_2$ if there exists a position u such that $w_1.u = w_2$. Given a set of positions P, the *prefix closure* of P is the set $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$. Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t|_w$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of *replacing the subterm* $t|_w$ by the term s.

A substitution σ is a mapping from variables to terms $\{X_1/t_1, \ldots, X_n/t_n\}$ such that $X_i\sigma = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq x_j$ if $i \neq j$), and $X\sigma = X$ for all other variables X. Given a substitution $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$, the *domain* of σ is the set $Dom(\sigma) = \{X_1, \ldots, X_n\}$. For any substitution σ and set of variables V, $\sigma_{\upharpoonright V}$ denotes the substitution obtained from σ by restricting its domain to V, (i.e., $\sigma_{\upharpoonright V}(X) = X\sigma$ if $X \in V$, otherwise $\sigma_{\upharpoonright V}(X) = X$). Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. By $match_s(t)$, we denote the function that returns a matcher of t in s if such a matcher exists, otherwise $match_s(t)$ returns *fail*.

We consider three different kinds of conditions that may appear in a conditional Maude theory: an *equational condition*[1] e is any (ordinary) equation $t = t'$, with $t, t' \in \tau(\Sigma, \mathcal{V})$; a *matching condition* is a pair $p := t$ with $p, t \in \tau(\Sigma, \mathcal{V})$; a *rewrite expression* is a pair $t \Rightarrow p$, with $p, t \in \tau(\Sigma, \mathcal{V})$.

2.2. Program Equations and Rules

A *conditional* equation is an expression of the form $\lambda = \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is either an equational condition, or a matching condition. When the condition C is empty, we simply write $\lambda = \rho$. A conditional equation $\lambda = \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible*, iff (i) $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{i=1}^{n} \mathcal{V}ar(c_i)$, and (ii) for each c_i, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is an equational condition, and $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is a matching condition $p := e$.

A *conditional* rule is an expression of the form $\lambda \to \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is an equational condition, a

[1] A boolean equational condition $b = true$, with $b \in \tau(\Sigma, \mathcal{V})$ of sort Bool is simply abbreviated as b. A *boolean condition* is a sequence of abbreviated boolean equational conditions.

```
mod BANK is inc INT .
  sorts Account Msg State Id .
  subsorts  Account Msg < State .
  var Id Id1 Id2 : Id .
  var b b1 b2 nb nb1 nb2 M : Int .
  op empty-state : -> State .
  op  _;_ : State State -> State [assoc comm id: empty-state] .
  op  ac : Id Int -> Account [ctor] .
  ops A B C D: Id .
  ops credit debit : Id Int -> Msg [ctor] .
  op  transfer : Id Id Int -> Msg  [ctor] .
  crl [credit] : ac(Id,b);credit(Id,M) => ac(Id,nB) if nB := b + M .
  crl [debit] : ac(Id,b);debit(Id,M) => ac(Id,nb) if b >= M /\ nb := b - M .
  crl [transfer] : ac(Id1,b1);ac(Id2,b2);transfer(Id1,Id2,M) => ac(Id1,nb1);ac(Id2,nb2)
                if ac(Id1,b1);debit(Id1,M) => ac(Id1,nb1) /\ ac(Id2,b2);credit(Id2,M) => ac(Id2,nb2) .
endm
```

Figure 1: Maude specification of a distributed banking system

matching condition, or a rewrite expression. When the condition C is empty, we simply write $\lambda \to \rho$. A conditional rule $\lambda \to \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible* iff it fulfils the exact analogy of the admissibility constraints (i) and (ii) for the equational conditions and the matching conditions, plus the following additional constraint: for each rewrite expression c_i in C of the form $e \Rightarrow p$, $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$.

The set of variables that occur in a (conditional) rule/equation r is denoted by $\mathcal{V}ar(r)$. Note that admissible equations and rules can contain extra-variables (i.e., variables that appear in the right-hand side or in the condition of a rule/equation but do not occur in the corresponding left-hand side). The admissibility requirements ensure that all the extra-variables will become instantiated whenever an admissible rule/equation is applied.

An *order-sorted equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of (oriented) admissible, conditional equations, and B is a collection of unconditional equational axioms (e.g., associativity, commutativity, and unity) that can be associated with any binary operator of Σ. The equational theory E induces a congruence relation on the term algebra $T(\Sigma, \mathcal{V})$, which is denoted by $=_E$. A *conditional rewrite theory* (or simply, rewrite theory) is a triple $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where $(\Sigma, \Delta \cup B)$ is an order-sorted equational theory, and R is a set of admissible conditional rules[2].

Example 2.1

Consider the Maude specification of Figure 1 that encodes a conditional rewrite theory modeling a simple, distributed banking system. Each state of the system is modeled as a multiset (i.e., an associative and commutative list) of elements of the form $\mathtt{e_1}; \mathtt{e_2}; \ldots\ldots; \mathtt{e_n}$. Each element $\mathtt{e_i}$ is either (i) a bank account `ac(Id,b)` where `Id` is

[2]Equational specifications in Maude can be theories in membership equational logic, which may include conditional membership axioms not addressed in this paper.

the owner of the account and `b` is the account balance; or (ii) a message modeling a debit, credit, or transfer operation. These account operations are implemented via three rewrite rules: namely, the `debit`, `credit`, and `transfer` rules.

Note that the rule `credit` contains the matching condition `nb := b + M`, while in the rule `debit` an equational condition `b >= M` and a matching condition `nb := b - M` occur. Finally, two rewrite expressions `ac(Id1,b1);debit(Id1,M) => ac(Id1,nb1)` and `ac(Id2,b2);credit(Id2,M) => ac(Id2,nb2)` appear in the rule `transfer`.

The conditional slicing technique formalized in this article is formulated by considering the precise way in which Maude proves the conditional rewriting steps, which we describe in the following section (see Section 5.2 in [7] for more details).

3. Conditional Rewriting Modulo Equational Theories

Given a conditional rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the conditional rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual conditional rewrite relation on terms [15] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [16], that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, in general, $\rightarrow_{R/E}$ is undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The Maude interpreter implements conditional rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$, that allow rules and equations to be intermixed in the rewriting process by simply using an algorithm of matching modulo B. We define $\rightarrow_{R\cup\Delta,B}$ as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of Δ (oriented from left to right) as simplification rules: thus, for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a term $t\downarrow_\Delta$ to which no further equations can be applied. The term $t\downarrow_\Delta$ is called a *canonical form* of t w.r.t. Δ. On the other hand, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [7].

Formally, $\rightarrow_{R,B}$ and $\rightarrow_{\Delta,B}$ are defined as follows. Given a rewrite rule $r = (\lambda \rightarrow \rho \textit{ if } C) \in R$ (resp., an equation $e = (\lambda = \rho \textit{ if } C) \in \Delta$), a substitution σ, a term t, and a position w of t, $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) iff $\lambda\sigma =_B t_{|w}$, $t' = t[\rho\sigma]_w$, and C *evaluates to true* w.r.t σ. When no confusion can arise, we simply write $t \rightarrow_{R,B} t'$ (resp. $t \rightarrow_{\Delta,B} t'$) instead of $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$).

Note that the evaluation of a condition C is typically a recursive process, since it may involve further (conditional) rewrites in order to normalize C to *true*. Specifically, an equational condition e *evaluates to true* w.r.t. σ if $e\sigma\downarrow_\Delta =_B \textit{true}$; a matching equation $p := t$ *evaluates to true* w.r.t. σ if $p\sigma =_B t\sigma\downarrow_\Delta$; a rewrite expression $t \Rightarrow p$ *evaluates*

to true w.r.t. σ if there exists a rewrite sequence $t\sigma \to^*_{R \cup \Delta, B} u$, such that $u =_B p\sigma$[3]. Although rewrite expressions and matching/equational conditions can be intermixed in any order, we assume that their satisfaction is attempted sequentially from left to right, as in Maude.

Under appropriate conditions on the rewrite theory, a rewrite step modulo E on a term t can be implemented without loss of completeness by applying the following rewrite strategy [17]: (i) reduce t w.r.t. $\to_{\Delta,B}$ until the canonical form $t\downarrow_\Delta$ is reached; (ii) rewrite $t\downarrow_\Delta$ w.r.t. $\to_{R,B}$.

An *execution trace* $\mathcal{T}$ in the rewrite theory $(\Sigma, \Delta \cup B, R)$ is a rewrite sequence

$$s_0 \to^*_{\Delta,B} s_0{\downarrow_\Delta} \to_{R,B} s_1 \to^*_{\Delta,B} s_1{\downarrow_\Delta} \dots$$

that interleaves $\to_{\Delta,B}$ rewrite steps and $\to_{R,B}$ steps following the strategy mentioned above.

Given an execution trace $\mathcal{T}$, it is always possible to expand $\mathcal{T}$ in an *instrumented* trace $\mathcal{T}'$ in which every application of the matching modulo B algorithm is mimicked by the explicit application of a suitable equational axiom, which is also oriented as a rewrite rule [9]. This way, any given instrumented execution trace consists of a sequence of (standard) rewrites using the conditional equations ($\to_\Delta$), conditional rules ($\to_R$), and axioms ($\to_B$).

Example 3.1

Consider the rewrite theory in Example 2.1 together with the following execution trace $\mathcal{T}$: `credit(A,2+3);ac(A,10)` $\to_{\Delta,B}$ `credit(A,5);ac(A,10)` $\to_{R,B}$ `ac(A,10)` Thus, the corresponding instrumented execution trace is given by expanding the commutative "step" applied to the term `credit(A,5);ac(A,10)` using the implicit rule (`X;Y` $\to$ `Y;X`) in B that models the commutativity axiom for the (juxtaposition) operator `_;_`.

`credit(A,2+3);ac(A,10)` $\to_\Delta$`credit(A,5);ac(A,10)` $\to_B$`ac(A,10);credit(A,5)` $\to_R$`ac(A,15)`

Also, typically hidden inside the B-matching algorithms, some transformations allow terms that contain operators that obey associative-commutative axioms to be rewritten by first producing a single representative of their AC congruence class [9]. For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b, f(f(b,a),c)) =_B f(f(b,c), f(a,b))$, we can represent it by using the "internal sequence" of transformations $f(b, f(f(b,a),c)) \to^*_{\mathit{flat}_B} f(a,b,b,c) \to^*_{\mathit{unflat}_B} f(f(b,c), f(a,b))$, where the first transformation corresponds to a

[3]Technically, to properly evaluate a rewrite expression $t \Rightarrow p$ or a matching condition $p := t$, the term p is required to be a Δ-pattern —i.e., a term p such that, for every substitution σ, if $x\sigma$ is a canonical form w.r.t. Δ for every $x \in \mathit{Dom}(\sigma)$, then $p\sigma$ is also a canonical form w.r.t. Δ.

```
mod M is inc NAT .
 var X  : Nat .
 var Y : NzNat .
 op _mod_ : Nat NzNat -> Nat .
 ceq X mod Y = X if Y > X .
 ceq X mod Y = (X - Y) mod Y
       if Y <= X .
endm
```

Figure 2: The `_mod_` operator

flattening transformation sequence that obtains the AC canonical form, while the second transformation corresponds to the inverse, unflattening one.

In the sequel, we assume all execution traces are instrumented as explained above. By abuse of notation, we frequently denote the rewrite relations $\to_\Delta$, $\to_R$, $\to_B$ by $\to$. Also, we denote the transitive and reflexive (resp. transitive) closure of the relation $\to_\Delta \cup \to_R \cup \to_B$ by $\to^*$ (resp. $\to^+$).

4. Backward Conditional Slicing

A backward trace slicing methodology for RWL was first proposed in [9] that is only applicable to *unconditional* RWL theories, and, hence, it cannot be employed when the source program includes conditional equations and/or rules since it would deliver incorrect and/or incomplete trace slices. The following example illustrates why conditions cannot be disregarded by the slicing process.

Example 4.1

Consider the Maude specification in Figure 2, which computes the remainder of the division of two natural numbers, and the associated execution trace $\mathcal{T}$: `4 mod 5` $\to$ `4`. Assume that we are interested in observing the origins of the target symbol `4` that appears in the final state. If we disregard the condition `Y` $>$ `X` of the first conditional equation, the slicing technique of [9] computes the trace slice $\mathcal{T}^\bullet$: `4 mod` $\bullet \to$ `4`, which is not correct since there exist concrete instances of `4 mod` $\bullet$ that cannot be rewritten to `4` using the first rule —e.g., `4 mod 3` $\not\to$ `4`. By contrast, our novel conditional approach not only produces a trace slice, but also delivers a boolean condition that establishes the valid instantiations of the input term that generate the observed data. In this specific case, our slicing technique would deliver the pair [`4 mod` $\bullet \to$ `4`, $\bullet >$ `4`].

In this section, we formulate our conditional slicing algorithm for RWL computations. The algorithm is formalized by means of a transition system that traverses the execution traces from back to front. The transition system is given by a single inference rule that relies on a *backward rewrite step slicing* procedure that is based on a substitution refinement.

4.1. Term slices and term slice concretizations

A term slice of a term t is a term abstraction that disregards part of the information in t, that is, the irrelevant data in t are simply replaced by special $\bullet$-variables, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$, which are generated by calling the auxiliary function $fresh^{\bullet}$[4]. More formally, a term slice is defined as follows.

Definition 4.2 (term slice) *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term, and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. A* term slice *of t w.r.t. P is defined as follows:*

$$slice(t, P) = rslice(t, P, \Lambda), \text{ where}$$

$$rslice(t, P, p) = \begin{cases} f(rslice(t_1, P, p.1), \ldots, rslice(t_n, P, p.n)) & \\ & \text{if } t{=}f(t_1, \ldots, t_n) \text{ and } p \in \bar{P} \\ x & \text{if } t{=}x \text{ and } x \in \mathcal{V} \text{ and } p \in \bar{P} \\ fresh^{\bullet} & \text{otherwise} \end{cases}$$

When P is understood, a term slice of t w.r.t. P is simply denoted by $t^{\bullet}$.

Roughly speaking, a term slice t w.r.t. a set of positions P includes all symbols of t that occur within the paths from the root to any position in P, while each maximal subterm $t_{|p}$, with $p \notin P$, is abstracted by means of a $\bullet$-variable.

Given a term slice $t^{\bullet}$, a *meaningful* position p of $t^{\bullet}$ is a position $p \in \mathcal{P}os(t^{\bullet})$ such that $t^{\bullet}_{|p} \neq \bullet_i$, for some $i = 0, 1, \ldots$. By $\mathcal{MP}os(t^{\bullet})$, we denote the set that contains all the meaningful positions of $t^{\bullet}$. Symbols that occur at meaningful positions are called *meaningful* symbols.

Example 4.3

Let $t = d(f(g(a, h(b)), c), a)$ be a term, and let $P = \{1.1,\ 1.2\}$ be a set of positions of t. By applying Definition 4.2, we get the term slice $t^{\bullet} = slice(t, P) = d(f(g(\bullet_1, \bullet_2), c), \bullet_3)$ and the set of meaningful positions $\mathcal{MP}os(t^{\bullet}) = \{\Lambda,\ 1,\ 1.1,\ 1.2\}$.

Now we show how we particularize a term slice, i.e., we instantiate $\bullet$-variables with data that satisfy a given boolean condition that we call *compatibility* condition. Term slice concretization is formally defined as follows.

Definition 4.4 (term slice concretization) *Let $t, t' \in \tau(\Sigma, \mathcal{V})$ be two terms. Let $t^{\bullet}$ be a term slice of t and let $B^{\bullet}$ be a boolean condition. We say that t' is a* concretization *of $t^{\bullet}$ that is compatible with $B^{\bullet}$ (in symbols $t^{\bullet} \propto^{B^{\bullet}} t'$), if (i) there exists a substitution σ such that $t^{\bullet}\sigma = t'$, and (ii) $B^{\bullet}\sigma$ evaluates to true.*

[4]Each invocation of $fresh^{\bullet}$ returns a (fresh) variable $\bullet_i$ of appropriate sort, which is distinct from any previously generated variable $\bullet_j$.

Example 4.5

Let $t^\bullet = \bullet_1 + \bullet_2 + \bullet_2$ and $B^\bullet = (\bullet_1 > 6 \wedge \bullet_2 \leq 7)$. Then, $10+2+2$ is a concretization of $t^\bullet$ that is compatible with $B^\bullet$, while $4+2+2$ is not.

In the following, we formulate a backward trace slicing algorithm that, given an execution trace $\mathcal{T} : s_0 \to^* s_n$ and a term slice $s_n^\bullet$ of s_n, generates the sliced counterpart $\mathcal{T}^\bullet : s_0^\bullet \to^* s_n^\bullet$ of $\mathcal{T}$ that only encodes the information required to reproduce (the meaningful symbols of) the term slice $s_n^\bullet$. Additionally, the algorithm returns a companion compatibility condition $B^\bullet$ that guarantees the soundness of the generated trace slice.

4.2. Backward Slicing for Execution Traces

Consider an execution trace $\mathcal{T} : s_0 \to^* s_n$. A trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is defined w.r.t. a *slicing criterion* — i.e., a set of positions $\mathcal{O}_{s_n} \subseteq \mathcal{P}os(s_n)$ that refer to those symbols of s_n that we want to observe. Basically, the trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is obtained by removing all the information from $\mathcal{T}$ that is not required to produce the term slice $s_n^\bullet = slice(s_n, \mathcal{O}_{s_n})$. A trace slice is formally defined as follows.

Definition 4.6 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and let $\mathcal{T} : s_0 \stackrel{r_1,\sigma_1,w_1}{\to} s_1 \stackrel{r_2,\sigma_2,w_2}{\to} \ldots \stackrel{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. A* trace slice *of $\mathcal{T}$ w.r.t. $\mathcal{O}_{s_n}$ is a pair $[s_0^\bullet \to s_1^\bullet \to \ldots \to s_n^\bullet,\ B^\bullet]$, where*

1. *$s_i^\bullet$ is a term slice of s_i; for $i = 0, \ldots, n$, and $B^\bullet$ is a boolean condition;*
2. *$s_n^\bullet = slice(s_n, \mathcal{O}_{s_n})$;*
3. *for every term s'_0 such that $s_0^\bullet \propto^{B^\bullet} s'_0$, there exists an execution trace $s'_0 \to s'_1 \to \ldots \to s'_n$ in $\mathcal{R}$ such that*

   i) $s'_i \to s'_{i+1}$ is either the rewrite step $s'_i \stackrel{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to} s'_{i+1}$ or $s'_i = s'_{i+1}$, $i = 0, \ldots, n-1$;

   ii) $s_i^\bullet \propto^{B^\bullet} s'_i$, $i = 1, \ldots, n$.

Note that Point 3 of Definition 4.6 ensures that the rules involved in the sliced steps of $\mathcal{T}^\bullet$ can be applied again, at the corresponding positions, to every concrete trace $\mathcal{T}'$ that can be obtained by instantiating all the $\bullet$-variables in $s_0^\bullet$ with arbitrary terms. The following example illustrates the slicing of an execution trace.

Example 4.7

Consider the Maude specification of Example 2.1 together with the following execution trace

$\mathcal{T}$: `ac(A,30);debit(A,5);credit(A,3)` $\stackrel{\texttt{debit}}{\to}$ `ac(A,25);credit(A,3)` $\stackrel{\texttt{credit}}{\to}$ `ac(A,28)`.

Let `ac(A,|`$\bullet_1$`)` be a term slice of `ac(A,28)` generated with the slicing criterion $\{1\}$ — i.e., `ac(A,`$\bullet_1$`)`$=$*slice*(`ac(A,28)`, $\{1\}$). Then, the trace slice for $\mathcal{T}$ is $[\mathcal{T}^\bullet,\ \bullet_8 \geq \bullet_9]$ where $\mathcal{T}^\bullet$ is as follows

$$\texttt{ac(A,}\bullet_8\texttt{);debit(A,}\bullet_9\texttt{);credit(A,|}\bullet_4\texttt{)}\stackrel{\texttt{debit}}{\rightarrow}\texttt{ac(A,|}\bullet_3\texttt{);credit(A,}\bullet_4\texttt{)}\stackrel{\texttt{credit}}{\rightarrow}\texttt{ac(A,}\bullet_1\texttt{)}$$

Note that $\mathcal{T}^\bullet$ needs to be endowed with the compatibility condition $\bullet_8 \geq \bullet_9$ in order to ensure the applicability of the `debit` rule. In other words, any instance $s^\bullet\sigma$ of `ac(A,`$\bullet_8$`);debit(A,`$\bullet_9$`)` can be rewritten by the `debit` rule only if $\bullet_8\sigma \geq \bullet_9\sigma$.

Informally, given a slicing criterion $\mathcal{O}_{s_n}$ for the execution trace $\mathcal{T} = s_0 \rightarrow^* s_n$, at each rewrite step $s_{i-1} \rightarrow s_i$, $i = n, \ldots, 1$, our technique inductively computes the association between the meaningful information of s_i and the meaningful information in s_{i-1}. For each such rewrite step, the conditions of the applied rule are recursively processed in order to ascertain from s_i the meaningful information in s_{i-1}, together with the accumulated condition $B_i^\bullet$. The technique proceeds backwards, from the final term s_n to the initial term s_0. A simplified trace is obtained where each s_i is replaced by the corresponding term slice $s_i^\bullet$.

We define a transition system $(Conf,\ \bullet\!\!\rightarrow)$ [18] where *Conf* is a set of *configurations* and $\bullet\!\!\rightarrow$ is the transition relation that implements the backward trace slicing algorithm. Configurations are formally defined as follows.

Definition 4.8 *A configuration, written as* $\langle \mathcal{T},\ S^\bullet,\ B^\bullet \rangle$, *consists of three components:*
- *the execution trace* $\mathcal{T} : s_0 \rightarrow^* s_{i-1} \rightarrow s_i$ *to be sliced;*
- *the term slice* $s_i^\bullet$, *that records the computed term slice of* s_i
- *a boolean condition* $B^\bullet$.

The transition system $(Conf,\ \bullet\!\!\rightarrow)$ is defined as follows.

Definition 4.9 *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory, let* $\mathcal{T} = U \rightarrow^* W$ *be an execution trace in* $\mathcal{R}$, *and let* $V \rightarrow W$ *be a rewrite step. Let* $B_W^\bullet$ *and* $B_V^\bullet$ *be two boolean conditions, and let* $W^\bullet$ *be a term slice of* W. *Then, the transition relation* $\bullet\!\!\rightarrow \subseteq Conf \times Conf$ *is the smallest relation that satisfies the following rule:*

$$\frac{(V^\bullet,\ B_V^\bullet) = \textit{slice-step}(V \rightarrow W,\ W^\bullet,\ B_W^\bullet)}{\langle U \rightarrow^* V \rightarrow W,\ W^\bullet,\ B_W^\bullet \rangle \bullet\!\!\rightarrow \langle U \rightarrow^* V,\ V^\bullet,\ B_V^\bullet \rangle}$$

Roughly speaking, the relation $\bullet\!\!\rightarrow$ transforms a configuration $\langle U \rightarrow^* V \rightarrow W,\ W^\bullet,\ B_W^\bullet \rangle$ into a configuration $\langle U \rightarrow^* V,\ V^\bullet,\ B_V^\bullet \rangle$ by calling the function *slice-step*$(V \rightarrow W,\ W^\bullet,\ B_W^\bullet)$ of Section 4.3, which returns a rewrite step slice for $V \rightarrow W$. More precisely, *slice-step* computes a suitable term slice $V^\bullet$ of V and a boolean condition $B_V^\bullet$ that updates the compatibility condition specified by $B_W^\bullet$.

```
function slice-step(s --r,σ,w--> t, t•, B•_prev)
 1. if w ∉ MPos(t•)
 2.   then
 3.     s• = t•
 4.     B• = B•_prev
 5.   else
 6.     θ = {x/fresh• | x ∈ Var(r)}
 7.     ρ• = slice(ρ, MPos(t•_|w))
 8.     ψ_ρ = ⟨|θ, match_{ρ•θ}(t•_|w)|⟩
 9.     for i = n downto 1 do
10.       (ψ_i, B•_i) = process-condition(c_i, σ,
                                ⟨|ψ_ρ, ψ_n...ψ_{i+1}|⟩)
11.     od
12.     s• = t•[λ⟨|ψ_ρ, ψ_n...ψ_1|⟩]_w
13.     B• = (B•_prev ∧ B•_n... ∧ B•_1)(ψ_1 ψ_2 ... ψ_n)
14. fi
15. return (s•, B•)
```

Figure 3: Backward step slicing function.

The initial configuration $\langle s_0 \to^* s_n,\ slice(s_n, \mathcal{O}_{s_n}),\ true\rangle$ is transformed until a terminal configuration $\langle s_0,\ s_0^\bullet,\ B_0^\bullet\rangle$ is reached. Then, the computed trace slice is obtained by replacing each term s_i by the corresponding term slice $s_i^\bullet$, $i = 0, \ldots, n$, in the original execution trace $s_0 \to^* s_n$. The algorithm additionally returns the accumulated compatibility condition $B_0^\bullet$ attained in the terminal configuration.

More formally, the backward trace slicing of an execution trace w.r.t. a slicing criterion is implemented by the function *backward-slicing* defined as follows.

Definition 4.10 (Backward trace slicing algorithm) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory, and let $\mathcal{T} : s_0 \to^* s_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. Then, the function backward-slicing is computed as follows:*

$$backward\text{-}slicing(s_0 \to^* s_n, \mathcal{O}_{s_n}) = [s_0^\bullet \to^* s_n^\bullet,\ B_0^\bullet]$$

iff there exists a transition sequence in $(Conf,\ \bullet\!\!\to)$

$$\langle s_0 \to^* s_n,\ s_n^\bullet,\ true\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s_{n-1}^\bullet,\ B_{n-1}^\bullet\rangle \bullet\!\!\to^* \langle s_0,\ s_0^\bullet,\ B_0^\bullet\rangle$$
$$where\ s_n^\bullet = slice(s_n,\ \mathcal{O}_{s_n})$$

In the following, we formulate the auxiliary procedure for the slicing of conditional rewrite steps.

4.3. The function slice-step

The function *slice-step*, which is outlined in Figure 3, takes three parameters as input (a rewrite step $\mu : s \overset{r,\sigma,w}{\to} t$ (with $r = \lambda \to \rho\ if\ C$[5]), a term slice $t^\bullet$ of t, and a

[5]Since equations and axioms are both interpreted as rewrite rules in our formulation, we often abuse the notation $\lambda \to \rho\ if\ C$ to denote rules as well as (oriented) equations and axioms.

compatibility condition $B^\bullet_{prev}$) and delivers the term slice $s^\bullet$ and a new compatibility condition $B^\bullet$. Within the algorithm *slice-step*, we use an auxiliary operator $\langle\!|\sigma_1, \sigma_2|\!\rangle$ that refines (overrides) a substitution σ_1 with a substitution σ_2, where both σ_1 and σ_2 may contain $\bullet$-variables. The main idea behind $\langle\!|_,_|\!\rangle$ is that, for the slicing of the step μ, all variables in the applied rewrite rule r are naïvely assumed to be initially bound to irrelevant data $\bullet$, and the bindings are incrementally refined as we (partially) solve the conditions of r.

Definition 4.11 (refinement) *Let σ_1 and σ_2 be two substitutions. The refinement of σ_1 w.r.t. σ_2 is defined by the operator $\langle\!|_,_|\!\rangle$ as follows:*

$$\langle\!|\sigma_1, \sigma_2|\!\rangle = \sigma_{\restriction Dom(\sigma_1)},\ \textit{where}$$

$$x\sigma = \begin{cases} x\sigma_2 & \textit{if } x \in Dom(\sigma_1) \cap Dom(\sigma_2) \\ x\sigma_1\sigma_2 & \textit{if } x \in Dom(\sigma_1) \setminus Dom(\sigma_2) \wedge \sigma_2 \neq \textit{fail} \\ x\sigma_1 & \textit{otherwise} \end{cases}$$

Note that $\langle\!|\sigma_1, \sigma_2|\!\rangle$ differs from the (standard) instantiation of σ_1 with σ_2. We write $\langle\!|\sigma_1, .., \sigma_n|\!\rangle$ as a compact denotation for $\langle\!|\langle\!|..\langle\!|\sigma_1, \sigma_2|\!\rangle, .., \sigma_{n-1}|\!\rangle, \sigma_n|\!\rangle$.

Example 4.12

Let $\sigma_1 = \{x/\bullet_1,\ y/\bullet_2\}$ and $\sigma_2 = \{x/a,\ \bullet_2/g(\bullet_3),\ z/5\}$ be two substitutions. Thus, $\langle\!|\sigma_1, \sigma_2|\!\rangle = \{x/a,\ y/g(\bullet_3)\}$.

Roughly speaking, the function *slice-step* works as follows. When the rewrite step μ occurs at a position w that is not a meaningful position of $t^\bullet$ (in symbols, $w \notin \mathcal{MP}os(t^\bullet)$), trivially μ does not contribute to producing the meaningful symbols of $t^\bullet$. Therefore, the function returns $s^\bullet = t^\bullet$, with the input compatibility condition $B^\bullet_{prev}$.

Example 4.13

Consider the Maude specification of Example 2.1 and the following rewrite step μ: `ac(A,30);debit(A,5);credit(A,3)` $\stackrel{\texttt{debit}}{\to}$ `ac(A,25);credit(A,3)`. Let $\bullet_1;\texttt{credit}(\texttt{A},3)$ be a term slice of `ac(A,25);credit(A,3)`. Since the rewrite step μ occurs at position $1 \notin \mathcal{MP}os(\bullet_1;\texttt{credit}(\texttt{A},3))$, the term `ac(A,25)` introduced by μ in `ac(A,25);credit(A,3)` is completely ignored in $\bullet_1;\texttt{credit}(\texttt{A},3)$. Hence, the computed term slice for

$$\texttt{ac}(\texttt{A},30);\texttt{debit}(\texttt{A},5);\texttt{credit}(\texttt{A},3)$$

is the very same $\bullet_1;\texttt{credit}(\texttt{A},3)$.

On the other hand, when $w \in \mathcal{MP}os(t^\bullet)$, the computation of $s^\bullet$ and $B^\bullet$ involves a more in-depth analysis of the rewrite step, which is based on an inductive refinement process that is obtained by recursively processing the conditions of the applied rule. More specifically, we initially define the substitution $\theta = \{x/\mathit{fresh}^\bullet \mid x \in Var(r)\}$ that binds each variable in r to a fresh $\bullet$-variable. This corresponds to assuming that all the information in μ, which is introduced by the substitution σ, can be marked as irrelevant. Then, θ is incrementally refined using the following two-step procedure.

Step 1. We compute the matcher $match_{\rho\theta}(t^\bullet_{|w})$ and then generate the refinement ψ_ρ of θ w.r.t. $match_{\rho\theta}(t^\bullet_{|w})$ (in symbols, $\psi_\rho = \langle\!| \theta, match_{\rho\theta}(t^\bullet_{|w}) |\!\rangle$). Roughly speaking, the refinement ψ_ρ updates the bindings of θ with the meaningful information extracted from $t^\bullet_{|w}$.

Example 4.14

Consider the rewrite theory in Example 2.1 together with the following rewrite step $\mu_{\texttt{debit}}$: `ac(A,30);debit(A,5)` $\stackrel{\texttt{debit}}{\to}$ `ac(A,25)` that involves the application of the `debit` rule whose right-hand side is $\rho_{\texttt{debit}}$ =`ac(Id,nb)`. Let $t^\bullet$ =`ac(A,`$\bullet_1$`)` be a term slice of `ac(A,25)`. Then, the initially ascertained substitution for μ is

$$\theta = \{\texttt{Id}/\bullet_2, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_5\}$$

and $match_{\rho_{\texttt{debit}}\theta}(t^\bullet) = match_{\texttt{ac(}\bullet_2,\bullet_5\texttt{)}}(\texttt{ac(A,}\bullet_1\texttt{)}) = \{\bullet_2/\texttt{A},\ \ \bullet_5\ /\bullet_1\}$. Thus, the substitution

$$\psi_{\rho_{\texttt{debit}}} = \langle\!| \theta, \psi_{\rho_{\texttt{debit}}} |\!\rangle = \{\texttt{Id/A},\ \texttt{b}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{nb}/\bullet_1\}.$$

That is, $\psi_{\rho_{\texttt{debit}}}$ refines θ by replacing the uninformed binding $\texttt{Id}/\bullet_2$, with `Id/A`.

Step 2. Let $C\sigma = c_1\sigma \wedge \ldots \wedge c_n\sigma$ be the instance of the condition in the rule r that enables the rewrite step μ. We process each (sub)condition $c_i\sigma$, $i = 1, \ldots, n$, in reversed evaluation order, i.e., from $c_n\sigma$ to $c_1\sigma$, by using the auxiliary function *process-condition* given in Figure 4 that generates a pair $(\psi_i,\ B^\bullet_i)$ such that ψ_i is used to further refine the partially ascertained substitution $\langle\!| \psi_\rho, \psi_n, \ldots, \psi_{i+1} |\!\rangle$ that is computed by incrementally analyzing conditions $c_n\sigma, c_{n-1}\sigma \ldots, c_{i+1}\sigma$, and $B^\bullet_i$ is a boolean condition that is derived from the analysis of the condition c_i.

When the whole $C\sigma$ has been processed, we get the refinement $\langle\!| \psi_\rho, \psi_n, ..., \psi_1 |\!\rangle$, which basically encodes all the instantiations required to construct the term slice $s^\bullet$ from $t^\bullet$. More specifically, $s^\bullet$ is obtained from $t^\bullet$ by replacing the subterm $t^\bullet_{|w}$ with the left-hand side λ of r instantiated with $\langle\!| \psi_\rho, \psi_n, \ldots, \psi_1 |\!\rangle$. Furthermore, $B^\bullet$ is built by collecting all the boolean compatibility conditions $B^\bullet_i$ delivered by *process-condition* and instantiating them with the composition of the computed refinements

function *process-condition*$(c,\ \sigma,\ \theta)$
1. **case** c **of**
2. $(p := t) \vee (t \Rightarrow p)$ **:**
3. **if** $(t\sigma = p\sigma)$
4. **then return** $(\{\},\ true)$ **fi**
5. $Q = \mathcal{MP}os(p\theta)$
6. $[t^{\bullet} \to^{*} p^{\bullet},\ B^{\bullet}] =$ $backward\text{-}slicing(t\sigma \to^{*} p\sigma,\ Q)$
7. $t^{\bullet\prime} = slice(t, \mathcal{MP}os(t^{\bullet}))$
8. $\psi = match_{t^{\bullet\prime}\theta}(t^{\bullet})$
9. e **:**
10. $\psi = \{\ \}$
12. $B^{\bullet} = e\theta$
12. **end case**
13. **return** $(\psi,\ B^{\bullet})$

Figure 4: Condition processing function.

$\psi_1 \ldots \psi_n$. It is worth noting that *process-condition* handles rewrite expressions, equational conditions, and matching conditions differently. More specifically, the pair (ψ_i, B_i) that is returned after processing each condition c_i is computed as follows.

– **Matching conditions.** Let c be a matching condition with the form $p := m$ in the condition of rule r. During the execution of the step $\mu : s \overset{r,\sigma,w}{\to} t$, recall that c is evaluated as follows: first, $m\sigma$ is reduced to its canonical form $m\sigma\downarrow_{\Delta}$, and then the condition $m\sigma\downarrow_{\Delta} =_B p\sigma$ is checked. Therefore, the analysis of the matching condition $p := m$ during the slicing process of μ implies slicing the (internal) execution trace $\mathcal{T}_{int} = m\sigma \to^{*} p\sigma$, which is done by recursively invoking the function *backward-slicing* for execution trace slicing with respect to the meaningful positions of the term slice $p\theta$ of p, where θ is a refinement that records the meaningful information computed so far. That is, $[m^{\bullet} \to^{*} p^{\bullet},\ B^{\bullet}] = backward\text{-}slicing(m\sigma \to^{*} p\sigma,\ \mathcal{MP}os(p\theta))$. The result delivered by the function *backward-slicing* is a trace slice $m^{\bullet} \to^{*} p^{\bullet}$ with compatibility condition $B^{\bullet}$.

  In order to deliver the final outcome for the matching condition $p := m$, we first compute the substitution $\psi = match_{m\theta}(m^{\bullet})$, which is the substitution needed to refine θ, and then the pair $(\psi,\ B^{\bullet})$ is returned.

Example 4.15

Consider the rewrite step $\mu_{\texttt{debit}}$ of Example 4.14 together with the refined substitution $\theta = \{\texttt{Id/A},\ \texttt{b}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{nb}/\bullet_1\}$. We process the condition

```
nb := b - M
```

of `debit` by considering an internal execution trace $\mathcal{T}_{int} = \texttt{30} - \texttt{5} \to \texttt{25}$ [6]. By invoking the *backward-slicing* function, the trace slice result is $[\bullet_6 \to \bullet_6,\ true]$. The final outcome is given by $match_{\bullet_7 - \bullet_8}(\bullet_6)$, which is $fail$. Thus θ does not need any further refinement.

– **Rewrite expressions.** The case when c is a rewrite expression $t \Rightarrow p$ is handled similarly to the case of a matching equation $p := t$, with the difference that t can be reduced by using the rules of R in addition to equations and axioms.

– **Equational conditions.** During the execution of the rewrite step $\mu : s \stackrel{r,\sigma,w}{\to} t$, the instance $e\sigma$ of an equational condition e in the condition of the rule r is just fulfilled or falsified, but it does not bring any instantiation into the output term t. Therefore, when processing $e\sigma$, no meaningful information to further refine the partially ascertained substitution θ must be added. However, the equational condition e must be recorded in order to compute the compatibility condition $B^\bullet$ for the considered conditional rewrite step. In other words, after processing an equational condition e, we deliver the tuple $(\psi,\ B^\bullet)$, with $\psi = \{\ \}$ and $B^\bullet = e\theta$. Note that the condition e is instantiated with the updated substitution θ, in order to transfer only the meaningful information of $e\sigma$ computed so far in e.

Example 4.16

Consider the refined substitution given in Example 4.15

$$\theta = \{\texttt{Id/A},\ \texttt{b}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{nb}/\bullet_1\}$$

together with the rewrite step $\mu_{\texttt{debit}}$ of Example 4.14 that involves the application of the `debit` rule. After processing the condition `b >= M` of `debit`, we deliver $B^\bullet = (\bullet_3 >= \bullet_4)$.

The correctness of our backward conditional slicing technique is established in [10]. In order to prove this result, we first demonstrate an auxiliary lemma that establishes the correctness of the *backward-slicing* procedure for a single rewrite step. Then we use this result to prove the correctness of *backward-slicing* for a generic execution trace.

Lemma 4.17 *Let $\mathcal{R}$ be a rewrite theory. Let $\mathcal{T} : s \stackrel{r,\sigma,w}{\to} t$ be a (one-step) execution trace in the rewrite theory $\mathcal{R}$, and let $\mathcal{O}_t$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s^\bullet \to t^\bullet,\ B^\bullet]$ computed by backward-slicing$(\mathcal{T},\ \mathcal{O}_t)$ is a trace slice for $\mathcal{T}$.*

---

[6] Note that the trace `30-5`$\to$`25` involves an application of the Maude built-in operator "`-`". Given a built-in operator `op`, in order to handle the reduction `a op b` $\to$ `c` as an ordinary rewrite step, we add the rule `a op b` $\Rightarrow$ `c` to the considered rewrite theory.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Proof. (sketch) Given a rewrite step $\mu : s \stackrel{r,\sigma,w}{\rightarrow} t$, where $r = \lambda \rightarrow \rho \ if \ c_1 \wedge \ldots \wedge c_n$, the proof is an induction on the sum of the lengths of all the (internal) execution traces needed to prove the instantiated condition $(c_1 \wedge \ldots \wedge c_n)\sigma$. This way, each recursive call to *backward-slicing*, that is used to evaluate a given $c_i\sigma$, generates a trace slice by inductive hypothesis. Such trace slices are used, by the auxiliary function *slice-step*, to deliver a pair $(\psi_i,\ B_i^\bullet)$ for each c_i, where ψ_i records the relevant instantiations derived from the processing of the condition c_i, and $B_i^\bullet$ is the associated compatibility condition. Another key factor of the proof is that the inclusion of the redex pattern[7] of the rule r in $s^\bullet$ (i.e., $s^\bullet = t^\bullet[\lambda\langle\!|\psi_\rho, \psi_n...\psi_1|\!\rangle]_w$) guarantees the applicability of r to each concretization of $s^\bullet$ that is compatible with $B^\bullet$. ■

Theorem 4.18 (correctness) *[10]. Let $\mathcal{R}$ be a rewrite theory. Let $\mathcal{T} : s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow} \ldots \stackrel{r_n,\sigma_n,w_n}{\rightarrow} s_n$ be an execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing$(\mathcal{T},\ \mathcal{O}_{s_n})$ is a trace slice for $\mathcal{T}$.*

Proof. (sketch) The proof proceeds by induction on the length of $\mathcal{T} : s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow} \ldots \stackrel{r_n,\sigma_n,w_n}{\rightarrow} s_n$. Basically, by inductive hypothesis, we assume that $[s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B_1^\bullet]$ is the trace slice generated by invoking *backward-slicing* on the (sub)trace $s_1 \stackrel{r_2,\sigma_2,w_2}{\rightarrow} \ldots \stackrel{r_n,\sigma_n,w_n}{\rightarrow} s_n$ w.r.t. the slicing criterion $\mathcal{O}_{s_n}$. We then use Lemma 4.18 to determine the trace slice for the first step $s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow} s_1$ of $\mathcal{T}$ w.r.t. the slicing criterion induced by term slice $s_1^\bullet$ (i.e., $\mathcal{MP}os(s_1^\bullet)$). Finally, we prove the theorem by composing the two trace slices in the obvious way. ■

5. The Julienne system

The slicing methodology for conditional RWL computations developed so far has been implemented in the slicing tool Julienne. The slicing engine of Julienne is written in Maude and consists of about 170 Maude function definitions (approximately 1K lines of source code). Julienne also comes with an intuitive Web user interface that is based on the AJAX technology, which allows the slicing engine to be used through a Java Web service. The implementation uses Maude meta-level capabilities and is extremely efficient. Actually, the current version of Maude can do more than 3 million rewritings per second on state-of-the-art processors, and the Maude compiler can reach up to 15 million rewrites per second. Julienne generalizes and supersedes a previous unconditional slicer mentioned in [9].

The architecture of Julienne, which is depicted in Figure 5, consists of three system modules named **IT-Builder**, **Slicer**, and **Pretty-Printer**.

[7] A *redex pattern* is the nonvariable part of the left hand side of a rule. For example, consider the rule $r : f(g(x), a) \rightarrow h(x)$ such that f, g, a are operators and x is a variable; then, the redex pattern of r is the term $f(g(\square), a)$ where $\square$ is a hole that can eventually be filled in by virtue of a rewrite step.

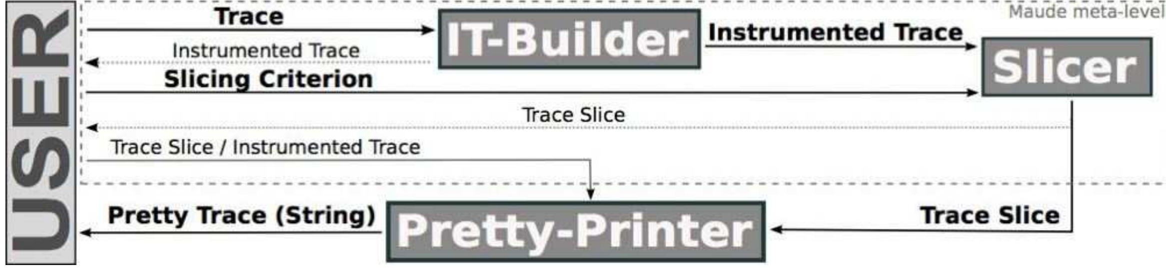


Figure 5: JULIENNE architecture

IT-Builder. The Instrumented Trace Builder (**IT-Builder**) module is a pre-processor that provides an expanded instrumented version of the original trace in which all reduction steps are explicitly represented, including equational simplification steps and applications of the *matching modulo* algorithm that are not explicitly recorded within the trace meta-representation given by the Maude meta-level operators. Showing all rewrites is not only required to successfully apply our trace slicing methodology in [10], but it can also be extremely useful for debugging purposes because it permits the user to inspect the equational simplification subcomputations that occur in a given trace.

Slicer. This module implements the trace slicing method by using Maude reflection and meta-level functionality. Specifically, it defines a new meta-level command called `back-sl` (*backward-slicing*) that takes as input an instrumented trace $t \to^* s$ (given as a Maude term of sort `Trace`) and a slicing criterion that represents the target symbols of the state s to be observed. It then delivers (i) a trace slice in which the data that are not relevant w.r.t. the chosen criterion are replaced by special •-variables, and (ii) a compatibility condition that ensures the correctness of the generated trace slice. This module is also endowed with a simple pattern-matching filtering language that helps to select the target symbols in s without the encumbrance of having to refer to them by their addressing positions. Roughly speaking, the slicing criterion is specified by a pattern p that consists of two wild cards (`?` and `_`) that are used to identify (resp. discard) the data of interest (resp. of no matter) inside s. Target symbols in s are then automatically retrieved by pattern matching s within p. For instance, the slicing criterion

$$\texttt{ac}(\texttt{B}, ?)\ ;\ _\ ;\ \texttt{ac}(\texttt{D}, ?)$$

is matched by the term

$$\texttt{ac}(\texttt{B}, 20)\ ;\ \texttt{ac}(\texttt{C}, 16)\ ;\ \texttt{ac}(\texttt{D}, 30)$$

and identifies as being relevant only the balance values of `B` and `D` (20 and 30, respectively), while any other data in the configuration (for `C`'s account) are disregarded.

Pretty-Printer. This module implements the command `prettyPrint`, which provides a human-readable, nicely structured view of the generated trace slice where the carried

```
mod BANK_ERR is inc INT .
  sorts Account Msg State Id .
  subsorts  Account Msg < State .
  var Id Id1 Id2 : Id .
  var b b1 b2 nb nb1 nb2 M : Int .
  op empty-state : -> State .
  op  _;_ : State State -> State [assoc comm id: empty-state] .
  op  ac : Id Int -> Account [ctor] .
  ops A B C D: Id .
  ops credit debit : Id Int -> Msg [ctor] .
  op  transfer : Id Id Int -> Msg  [ctor] .
  crl [credit]   :credit(Id,M);ac(Id,b) => ac(Id,nb) if nb := b+M .
  crl [debitERR] :debit(Id,M);ac(Id,b) => ac(Id,nb) if nb := b-M .
  crl [transfer] :transfer(Id1, Id2,M);ac(Id1,b1);ac(Id2,b2) => ac(Id1,nb1);ac(Id2,nb2)
   if debit(Id1,M);ac(Id1,b1) => ac(Id1,nb1) /\ credit(Id2,M);ac(Id2,b2) => ac(Id2,nb2) .
endm
```

Figure 6: Faulty Maude specification of a distributed banking system.

compatibility condition can be displayed or hidden, depending on the interest of the user. The pretty printer takes as input either a trace slice (typically the one delivered by the **Slicer** module) or any instrumented trace given by the user and delivers a pretty representation of the trace as a term of sort String that is aimed to favor better inspection and debugging activities within the Maude environment.

6. Julienne at work

In general, using conventional debuggers is an inefficient and time-consuming approach for understanding the program behavior, especially when a programmer is interested in observing only those parts of the program execution that relate to the incorrect output. In order to make program debugging and comprehension more efficient, it is important to focus the programmer's attention on the essential components (actions, states, equations and rules) of the program and their execution. Backward trace slicing provides a means to achieve this by pruning away the unrelated pieces of the computation.

6.1. Debugging Maude programs with Julienne

In debugging, one is often interested in analysing the specific execution of a program that exhibits anomalous behavior. However, the execution of Maude programs typically generates large and clumsy traces that are hard to browse and understand even when the programmer is assisted by tracing tools such as the Maude built-in tracing facility. This is because the tracer does not provide any means for identifying the contributing program parts of the program being debugged, and does not allow the programmer to distinguish related computations from unrelated computations. The inspection of these traces for debugging purposes is thus a cumbersome task that very often leads to no conclusion. In this scenario, backward trace slicing can play a meaningful role, since it can automatically reduce the size of the analyzed execution trace keeping track of

all and only those symbols that may cause the existence of an error or anomaly in the trace.

Basically, the idea is to feed the JULIENNE slicer with an execution trace $\mathcal{T}$ that represents a wrong behavior of a given Maude program, together with a slicing criterion that observes an erroneous outcome. The resulting trace slice is typically much smaller than the original one, since it only includes the information that is responsible for the production of the erroneous outcome. Thus, the programmer can easily navigate through the trace slice for program bugs to hunt. Let us see an example.

Example 6.1

Consider the Maude program `BANK_ERR` of Figure 6, which is a faulty mutation of the distributed banking system specified in Figure 1. More precisely, the rule `debit` has been replaced by the rule `debitERR` in which we have intentionally omitted the equational condition `M <= b`. Roughly speaking, the considered specification always authorizes withdrawals even in the erroneous case when the amount of money to be withdrawn is greater than the account balance.

Now let us consider an execution trace $\mathcal{T}_{bank}$ that starts in the initial state

```
ac(A,50) ; ac(B,20) ; ac(D,20) ; ac(C,20) ;
transfer(B,C,4) ; transfer(A,C,15) ;
debit(D,5) ; credit(A,10) ; transfer(A,D,20) ;
debit(C,50) ; credit(D,40)
```

and ends in the final state

```
ac(A,25) ; ac(B,16) ; ac(D,75) ; ac(C,-11)
```

We observe that the final state contains a negative balance for the client `C`, which is a clear symptom of malfunction of the `BANK_ERR` specification, since we assume that balances must be non-negative numbers. Therefore, we execute JULIENNE on the trace $\mathcal{T}_{bank}$ w.r.t. the slicing criterion specified by the pattern $\mathtt{ac}(\mathtt{C}, -\mathtt{11})$ in order to determine the cause of such a negative balance. The output delivered by JULIENNE is given in Table 1 and shows the (instrumented) input trace $\mathcal{T}_{bank}$ in the **Execution Trace** column, the simplified trace $\mathcal{T}^{\bullet}_{bank}$ in the **Sliced trace** column, the computed compatibility condition, and some other auxiliary data such as the size of the two traces, the reduction rate achieved, and the rules that have been applied in $\mathcal{T}_{bank}$ and $\mathcal{T}^{\bullet}_{bank}$.

It is worth noting that the trace slice $\mathcal{T}^{\bullet}_{bank}$ greatly simplifies the trace $\mathcal{T}_{bank}$ by deleting all the bank accounts and account operations that are not related to the client `C` as well as all the internal flat/unflat rewrite steps, which are needed to implement rewriting modulo associativity and commutativity. In fact, the computed reduction rate is 78%, which clearly shows the drastic pruning that we have obtained.

Now, a quick inspection of $\mathcal{T}^{\bullet}_{bank}$ allows the existence of a misbehaving account operation to be recognized. Specifically, state `12` in the trace slice $\mathcal{T}^{\bullet}_{bank}$ has been obtained by reducing the term $\mathtt{debit}(\mathtt{C}, \mathtt{50})$ by means of the rule `debit_ERR` even though the current

Step	RuleName	Execution trace	Sliced trace
1	'Start	ac(A,50) ; ac(B,20) ; ac(D,20) ; ac(C,20) ; credit(A,10) ; credit(D,40) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4)	ac(C,20) ; * ; * ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
2	unflattening	***deleted***	
3	credit	ac(B,20) ; ac(D,20) ; ac(C,20) ; credit(D,40) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(A,60)	ac(C,20) ; * ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
4/5	flattening / unflattening	***deleted***	
6	credit	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(D,60)	ac(C,20) ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
7/8	flattening / unflattening	***deleted***	
9	debitERR	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(D,55)	ac(C,20) ; * ; * ; * ; debit(C,50) ; transfer(*,C,4) ; transfer(*,C,15) ; *
10/11	flattening / unflattening	***deleted***	
12	debitERR	ac(A,60) ; ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(C,-30)	ac(C,-30) ; * ; * ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
13/14	flattening / unflattening	***deleted***	
15	transfer	ac(B,20) ; ac(C,-30) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(A,40) ; ac(D,75)	* ; ac(C,-30) ; * ; * ; transfer(*,C,4) ; transfer(*,C,15)
16/17	flattening / unflattening	***deleted***	
18	transfer	ac(B,20) ; ac(D,75) ; transfer(B,C,4) ; ac(A,25) ; ac(C,-15)	* ; * ; ac(C,-15) ; * ; transfer(*,C,4)
19/20	flattening / unflattening	***deleted***	
21	transfer	ac(A,25) ; ac(D,75) ; ac(B,16) ; ac(C,-11)	* ; * ; * ; ac(C,-11)
22	flattening	***deleted***	
		Condition:	TRUE
Total size:		1689	374
Reduction: 78%			

Table 1: JULIENNE output for the trace $\mathcal{T}_{bank}$ w.r.t. the criterion `ac(C,-11)`

balance of `C` was only `20`. This suggests to us that `debit_ERR` might be faulty since it does not conform to its intended semantics, which forbids any withdrawal greater than the current funds available.

To fully assess the practicability of our approach, we coupled JULIENNE with the Web-TLR system [5]. Web-TLR is a RWL-based tool, which is built on top of Maude, that allows real-size Web applications to be formally specified and verified by using the built-in Maude model-checker. In Web-TLR, a Web application is formalized by means of a Maude specification and then checked against a property specified in the *Linear Temporal Logic of Rewriting* (LTLR [19]), which is a temporal logic specifically designed to model-check rewrite theories. When a property is refuted by the LTLR model-checker, a counter-example in the form of a rewrite sequence that reveals an undesired, erroneous behavior is yielded. Backward trace slicing is then used to provide a simplified view of counter-examples which facilitates their inspection and debugging. A complete debugging session using backward trace slicing for the specification of a complex Webmail application was thoroughly described in [2].

```
mod MINMAX is inc INT .
   sorts List Pair .
   subsorts Nat < List .
   op _;_ : List List -> List [ctor assoc] .
   op PAIR : Nat Nat -> Pair .
   op 1st : Pair -> Nat .
   op 2nd : Pair -> Nat .
   op Max : Nat Nat -> Nat .
   op Min : Nat Nat -> Nat .
   op minmax : List -> Pair .
   var N X Y : Nat .
   var L : List .
   var P : Pair .
   crl [Max1] : Max(X,Y) => X if X >= Y .
   crl [Max2] : Max(X,Y) => Y if X < Y .
   crl [Min1] : Min(X,Y) => Y if X > Y .
   crl [Min2] : Min(X,Y) => X if X <= Y .
   rl [1st] : 1st(PAIR(X,Y)) => X .
   rl [2nd] : 2nd(PAIR(X,Y)) => Y .
   rl [minmax1] : minmax(N) =>  PAIR(N,N) .
   crl [minmax2] : minmax(N ; L) => PAIR(Min(N,1st(P)) , Max(N,2nd(P))) if P := minmax(L).
endm
```

Figure 7: Maude specification of the `minmax` function

6.2. *Trace querying with* Julienne

Execution traces of programs are a helpful source of information for program comprehension. However, they provide such a low-level picture of program execution that users may experience several difficulties in interpreting and analyzing them. Trace querying [20] allows a given execution trace to be analyzed at a higher level of abstraction by selecting only a subtrace of it that the user considers relevant.

Trace querying of Maude execution traces is naturally supported and completely automated by the trace slicer Julienne. Indeed, execution traces can be simply queried by providing a slicing criterion (in the form of a filtering pattern) that specifies the target symbols the user decides to monitor. Hence, backward trace slicing is performed w.r.t. the considered criterion to compute an abstract view (i.e., the trace slice) of the original execution trace that only includes the information that is strictly required to yield the target symbols under observation. This way, users can focus their attention on the monitored data, which might otherwise be overlooked in the concrete trace.

Moreover, backward trace slicing correctness provides, for free, a means to understand the program behavior w.r.t. partially-defined inputs since the computed compatibility condition constrains the possible values that non-relevant inputs (modeled by $\bullet$-variables) might assume. In other words, a trace slice $\mathcal{T}^\bullet$ can be thought of as an intensional representation of all the possible concrete traces, which are compatible concretizations of $\mathcal{T}^\bullet$, that lead to the production of the monitored target symbols.

Example 6.2

The Maude specification of Figure 7, inspired by a similar one in [21], specifies the operator `minmax` that takes as input a list of natural numbers `L` and computes a pair `(m,M)` where `m` (resp., `M`) is the minimum (resp., maximum) of `L`.

Step	Rule name	Execution trace	Sliced trace
1	'Start	minmax(4 ; 7 ; 0)	minmax(*23 ; *19 ; 0)
2	unflattening	minmax(4 ; 7 ; 0)	***deleted***
3	minmax2	PAIR(Min(4,1st(minmax(7 ; 0))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,1st(minmax(*19 ; 0))),Max(*1,*2))
4	minmax2	PAIR(Min(4,1st(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,1st(PAIR(Min(*8,1st(minmax(0))),*17))),Max(*1,*2))
5	1st	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,Min(*8,1st(minmax(0)))),Max(*1,*2))
6	minmax2	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))))	***deleted***
7	2nd	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,Max(7,2nd(minmax(0)))))	***deleted***
8	minmax1	PAIR(Min(4,Min(7,1st(PAIR(0,0)))),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,Min(*8,1st(PAIR(0,*11)))),Max(*1,*2))
9	1st	PAIR(Min(4,Min(7,0)),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,Min(*8,0)),Max(*1,*2))
10	Min1	PAIR(Min(4,0),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,0),Max(*1,*2))
11	Min1	PAIR(0,Max(4,Max(7,2nd(minmax(0)))))	PAIR(0,Max(*1,*2))
12	minmax1	PAIR(0,Max(4,Max(7,2nd(PAIR(0,0)))))	***deleted***
13	2nd	PAIR(0,Max(4,Max(7,0)))	***deleted***
14	Max1	PAIR(0,Max(4,7))	***deleted***
15	Max2	PAIR(0,7)	PAIR(0,*0)
		Condition:	*23 > 0 and *19 > 0
Total size:		679	312
Reduction: 54%			

Table 2: Julienne output for the trace $\mathcal{T}^{\bullet}_{minmax}$ w.r.t. the criterion $\mathtt{PAIR}(?,_)$

Let $\mathcal{T}_{minmax}$ be the execution trace that reduces the input term `minmax(4;7;0)` to the normal form `PAIR(0,7)`. Now, assume that we are only interested in analyzing the subtrace that generates `0` in `PAIR(0,7)` —i.e., the minimum of the list `4;7;0`.

Thus, we can query $\mathcal{T}_{minmax}$ by specifying the pattern `PAIR(?,_)` that allows us to trace back only the first argument of `PAIR`, while the second one is discarded. Julienne generates the trace slice $\mathcal{T}^{\bullet}_{minmax}$, that is given in Table 2, whose compatibility condition is $\mathtt{C}^{\bullet} = \bullet_{23} > 0 \wedge \bullet_{19} > 0$. The slice $\mathcal{T}^{\bullet}_{minmax}$ isolates all and only those function calls in the trace $\mathcal{T}$ that must be reduced to yield the minimum of the list `4;7;0`. Now, by analyzing the trace slice, it is immediate to see that the operators `2nd` and `Max` do not affect the observed result since they are not used in the trace slice. Also, by the correctness of our slicing technique, we can state that for every concrete instance $\mathtt{L_c}$ of the partially-defined input $\bullet_{23};\bullet_{19};0$ that meets the compatibility condition $\mathtt{C}^{\bullet}$, the minimum computed by the call $\mathtt{minmax}(\mathtt{L_c})$ will be `0`.

6.3. Dynamic program slicing

By running Julienne, we are able to obtain not only a more compact and focused trace that corresponds to the execution of the program, but also to uncover statement dependences that connect computationally related parts of the program. Hence a single walk of these dependences is sufficient to implement a form of dynamic program slicing.

Program slicing is the computation of the set of program statements, the program slice, that may affect the values at some point of interest. A program slice consists of a subset of the statements of the original program, sometimes with the additional constraint that a slice must constitute a syntactically valid, executable program. Relevant

applications of slicing include software maintenance, optimization, program analysis, and information flow control. An important distinction is that between static and dynamic slicing: whereas static slicing is performed with no other information than the source code itself, dynamic program slicing works on a specific execution of the program (i.e., a given execution trace) [22], hence it only reflects the actual dependences of that execution, resulting in smaller program slices than static ones. Dynamic slicing is usually achieved by dynamic data-flow analysis along the program execution trajectory. Although dynamic program slicing was first introduced to aid in user level debugging to locate sources of errors more easily, applications aimed at improving software quality, reliability, security and performance have also been identified as candidates for using dynamic program slicing.

Let us show how backward trace slicing can support the generation of dynamic program slices by detecting unused program rules in a given trace slice.

Example 6.3 ______

Consider the trace slice $\mathcal{T}^{\bullet}_{minmax}$ of Example 6.2 for the execution trace

$$\texttt{minmax}(4;7;0) \to^{*} \texttt{PAIR}(0,7)$$

w.r.t. the criterion $\texttt{PAIR}(?,_)$. Since operations `2nd` and `Max` are never used in $\mathcal{T}^{\bullet}_{minmax}$, we can generate a dynamic program slice of the `MINMAX` Maude module by deleting the rule definitions of `2nd` and `Max` and (possibly) replacing any function call to `2nd` or `Max` in the right-hand side or condition of the remaining rules by a special dumb constant $\square$ typed with an appropriate sort. Hence, the resulting program slice consists of the following rules:

```
crl [Min1] :  Min(X,Y) => Y if X > Y .
crl [Min2] :  Min(X,Y) => X if X <= Y .
rl  [1st] :  1st(PAIR(X,Y)) => X .
rl  [minmax1] :  minmax(N) => PAIR(N,N) .
crl [minmax2] :  minmax(N ; L) => PAIR(Min(N,1st(P)),□) if P := minmax(L).
```

______

6.4. Experimental evaluation

We have experimentally evaluated our tool in several case studies that are available at the Julienne Web site [23] and within the distribution package, which also contains a user guide, the source files of the slicer, and related literature.

To properly assess the performance and scalability, we have tested Julienne on several execution traces of increasing complexity: More precisely, we have considered

- two execution traces that model two runs of a fault-tolerant client-server communication protocol (`FTCP`) specified in Maude. Trace slicing has been performed according to two chosen criteria that aim at extracting information related to a

Example trace	Original trace size	Slicing criterion	Sliced trace size	*% reduction*
FTCP.$\mathcal{T}_1$	2054	FTCP.$\mathcal{T}_1.O_1$	294	85.69%
		FTCP.$\mathcal{T}_1.O_2$	316	84.62%
FTCP.$\mathcal{T}_2$	1286	FTCP.$\mathcal{T}_2.O_1$	135	89.40%
		FTCP.$\mathcal{T}_2.O_2$	97	92.46%
Maude-NPA.$\mathcal{T}_1$	21265	Maude-NPA.$\mathcal{T}_1.O_1$	2249	89.42%
		Maude-NPA.$\mathcal{T}_1.O_2$	2261	89.36%
Maude-NPA.$\mathcal{T}_2$	34681	Maude-NPA.$\mathcal{T}_2.O_1$	3015	91.30%
		Maude-NPA.$\mathcal{T}_2.O_2$	3192	90.79%
Web-TLR.$\mathcal{T}_1$	19114	Web-TLR.$\mathcal{T}_1.O_1$	3982	79.17%
		Web-TLR.$\mathcal{T}_1.O_2$	3091	83.83%
Web-TLR.$\mathcal{T}_2$	22018	Web-TLR.$\mathcal{T}_2.O_1$	2984	86.45%
		Web-TLR.$\mathcal{T}_2.O_2$	2508	88.61%
Web-TLR.$\mathcal{T}_3$	38983	Web-TLR.$\mathcal{T}_3.O_1$	2045	94.75%
		Web-TLR.$\mathcal{T}_3.O_2$	2778	92.87%
Web-TLR.$\mathcal{T}_4$	69491	Web-TLR.$\mathcal{T}_4.O_1$	8493	87.78%
		Web-TLR.$\mathcal{T}_4.O_2$	5034	92.76%

Table 3: Backward trace slicing benchmarks.

specific server and client in a scenario that involves multiple servers and clients, and tracking the response generated by the server according to a given client request.

- two execution traces generated by Maude-NPA [24], which is an RWL-based analysis tool for cryptographic protocols that takes into account many of the algebraic properties of cryptosystems. These include cancellation of encryption and decryption, Abelian groups (including exclusive-or), exponentiation, and homomorphic encryption. The considered traces model two instances of a well-known man-in-the-middle attack to the Needham-Schroeder network authentication protocol [25]. Specifically, they consist of a sequence of rewrite steps that represents the messages exchanged among three entities: an initiator A, a receiver B, and an intruder I that imitates A to establish a network session with B. The chosen slicing criteria selects the intruder's actions as well as the intruder's knowledge at each rewrite step discarding all the remaining session information.

- four counter-examples produced by model-checking a real-size Webmail application specified in Web-TLR. The chosen slicing criteria allowed several critical data to be isolated and inspected —e.g., the navigation of a malicious user, the messages exchanged by a specific Web browser with the Webmail server, and session data of interest (e.g., browser cookies).

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

The results of our experiments are shown in Table 3. The execution traces for the considered cases consist of sequences of 10–1000 states, each of which contains from 60 to 5000 characters. In all the experiments, the trace slices that we obtained show impressive reduction rates (up to ∼98%).

Other benchmark programs we have considered (e.g., the banking system specification of Figure 6 and a Maude program solving the famous crossing river puzzles) are available at the Julienne Web site. In most cases, the delivered trace slices were cleansed enough to be easily inspected by hand. Also, it is worth noting that that the slicer does not remove any information that is relevant, independently of the skills of the user.

With regard to the time required to perform the analyses, our implementation is extremely time efficient; the elapsed times are small even for very complex traces and scale linearly. For example, running the slicer for a 20Kb trace in a Maude specification with about 150 rules and equations –with AC rewrites– took less than 1 second (480.000 rewrites per second on standard hardware, 2.26GHz Intel Core 2 Duo with 4Gb of RAM memory).

7. Related Work

There are very few approaches that address the problem of tracing rewrite sequences in term rewrite systems [9, 12, 26, 27], and all of them apply to unconditional systems. The techniques in [9, 12, 26] rely on a labeling relation on symbols that allows data content to be traced back within the computation; this is achieved in [27] by formalizing a notion of dynamic dependence among symbols by means of contexts. In [12, 26], non-left linear and collapsing rules are not considered or are dealt with using ad-hoc strategies, while our approach requires no special treatment of such rules. Moreover, the first and only (unconditional) trace slicing technique that supports rules, equations, sorts, and algebraic axioms is [9].

The technique in [9] computes the reverse dependence among the symbols involved in an execution step by using a procedure (based on [26]) that dynamically labels the calls (terms) involved in the steps. In this paper, we describe a more general slicing technique that copes with conditional rewrite theories and simplifies the formal development in [9] by getting rid of the complex dynamic labeling algorithm that was needed to trace back the origins of the symbols of interest, replacing it with a simple mechanism for substitution refinement that allows control and data dependencies to be propagated between consecutive rewrite steps. Our technique also avoids manipulating the origins by recording their addressing positions; we simply and explicitly record the origins of the meaningful positions within the computed term slices themselves, without resorting to any other artifact.

We are not aware of any *trace slicer* that is comparable to Julienne for either imperative or declarative languages. To the best of our knowledge, there are just a couple of tools that only slightly relate to ours. Spyder [28] is a prototype debugger for C

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

that, thanks to the combination of dynamic *program slicing* and execution backtracking techniques, is able to automatically step-back, statement by statement, from any desired location in order to determine which statements in the program affect the value of an output variable for a given test case, and to determine the value of a given variable when the control last reached a given program location. In contrast to Spyder, our technique is based on *trace slicing* rather than *dynamic program slicing*, and needs much less storage to perform flow-back analysis, as it requires neither the construction of data and control dependence graphs nor the creation of an execution history.

The Haskell interactive debugger Hat [29] also allows execution traces to be explored backwards, starting from the program output or an error message (computation abort). Similarly to Spyder, this task is carried out by navigating a graph-like, supplementary data structure (redex trail) that records dependencies among function calls. Even if Hat is able to highlight the top-level "parent" function of any expression in the final trace state, it cannot be used to compose a full trace slice. Actually, at every point of the recreated trail, the function arguments are shown in fully evaluated form (the way they would appear at the end of the computation) even though, at the point at which they are shown, they would not yet have necessarily reached that form. A totally different, bytecode trace *compression* was proposed in [30] to help perform Java program analysis (e.g., dynamic *program slicing* [31]) over the compact representation.

8. Conclusions

We have developed a slicing transformation technique for rewriting logic programs that are written and executed in the Maude system. The technique can drastically reduce the size and complexity of the traces under examination. It is useful for execution trace analyses of sophisticated rewrite theories that may include conditional equations, equational axioms, and rules. An implementation of the proposed technique reveals that it does not come at the expense of performance with respect to existing Maude tools.This makes the tool attractive for Maude users, especially taking into account that program debugging and trace analysis in Maude is tedious with the current state of tools in its realm. The tool can be tuned to reveal all relevant information (including applied equation/rule, redex position, and matching substitution) for each single rewrite step that is obtained by (recursively) applying a conditional equation, algebraic axiom, or rule, which greatly improves the standard view of execution traces in Maude and their meta-representations. Moreover, it can provide both, a textual representation of the trace and its meta-level representation to be used for further automated analysis.

The Maude system currently supports two different approaches for debugging Maude programs: the internal debugger described in [14] (Chap. 22) and the declarative debugger of [32]. The first one is a traditional trace-based debugger that allows the user to define break points in the execution by selecting some operators or statements. When a break point is found, the debugger is entered, and the next rewrite is executed with tracing turned on. The user can also execute another Maude command, which in turn

can enter the (fully re-entrant) debugger. For large programs such repeated program executions may be very cumbersome. The declarative debugger of Maude is based on Shapiro's algorithmic debugging technique [33] and allows the debugging of wrong results (erroneous reductions, sort inferences, and rewrites) and incomplete results (not completely reduced normal forms, greater than expected least sorts, and incomplete sets of reachable terms) [32]. The debugging process starts from a computation considered incorrect by the user (typically from the initial term to an unexpected one) and locates a program fragment responsible for that error symptom. Then the debugger builds a debugging tree representing this computation and guides the user through it to find the bug, with several options to build, prune, and traverse the debugging tree. During the process, the system asks questions to an external oracle (generally the user or another program or formal specification) until a so-called buggy node is found, i.e., a node that contains an erroneous result but whose children have all correct results. Since a buggy node produces an erroneous output from correct inputs, it corresponds to an erroneous fragment of code that is pointed out as an error. Typical questions to the user have the form "Is it correct that term t rewrites (or fully reduces) to t'?" Since one of the main drawbacks of declarative debugging is the size of the debugging trees and the complexity of the questions to the oracle, the tool allows the debugging trees to be reduced in several ways (e.g., by considering as fully trusted code some statements and even whole modules). Sometimes, the user answers allow the tree to be further pruned and the corresponding questions referring to the nodes of the eliminated subtrees are consequently discarded. We think that our trace slicing technique can provide a complementary source of information to further shorten the declarative debugging process. By not considering some (sub-)computations that were proven by the trace slicer to have no influence on a criterion of interest, we might avoid many unnecessary questions to the user, which we plan to investigate as future work. In order to achieve this, we intend to develop a trace querying scheme for more sophisticated theories that may include membership axioms and to address the full integration of our tool within the formal tool environment of Maude. We also plan to explore other challenging applications of our trace slicing methodology, such as runtime verification [34] which is concerned with monitoring and analysis of system executions. Consider a programming language $\mathcal{L}$ which is given a RWL executable semantics. Then one can use the semantics as an interpreter to execute $\mathcal{L}$ programs (given as terms) directly within the semantics of their programming language as in [35], and hence Maude can be used to trace such executions. Then, by querying the trace slice w.r.t. a reference specification, runtime verification might be semantically grounded in our setting while it is commonly offhacked in more traditional approaches by means of program instrumentation.

References

[1] F. Chen, G. Rosu, Parametric trace slicing and monitoring, in: TACAS, volume 5505 of *LNCS*, Springer, 2009, pp. 246–261.

[2] M. Alpuente, D. Ballis, J. Espert, F. Frechina, D. Romero, Debugging of Web Applications with WEB-TLR, in: Proc. of 7th Int'l Workshop on Automated Specification and Verification of Web Systems WWV 2011, volume 61 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pp. 66–80.

[3] M. A. Francel, S. Rugaber, The Value of Slicing while Debugging, Sci. Comput. Program. 40 (2001) 151–169.

[4] M. Baggi, D. Ballis, M. Falaschi, Quantitative Pathway Logic for Computational Biology, in: Proc. of 7th Int'l Conference on Computational Methods in Systems Biology (CMSB '09), volume 5688 of *LNCS*, Springer, 2009, pp. 68–82.

[5] M. Alpuente, D. Ballis, J. Espert, D. Romero, Model-checking Web Applications with Web-TLR, in: Proc. of 8th Int'l Symposium on Automated Technology for Verification and Analysis (ATVA 2010), volume 6252 of *LNCS*, Springer, 2010, pp. 341–346.

[6] M. Alpuente, D. Ballis, D. Romero, Specification and Verification of Web Applications in Rewriting Logic, in: Formal Methods, Second World Congress FM 2009, volume 5850 of *LNCS*, Springer, 2009, pp. 790–805.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talco, Maude Manual (Version 2.6), Technical Report, SRI Int'l Computer Science Laboratory, 2011. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.

[8] N-Martí-Oliet, M. Palomino, A. Verdejo, Rewriting logic bibliography by topic: 1990-2011, Journal of Logic and Algebraic Prohramming (2012). To appear.

[9] M. Alpuente, D. Ballis, J. Espert, D. Romero, Backward Trace Slicing for Rewriting Logic Theories, in: Proc. of 23rd Int'l Conference on Automated Deduction CADE 2011, volume 6803 of *LNCS/LNAI*, Springer, 2011, pp. 34–48.

[10] M. Alpuente, D. Ballis, F. Frechina, D. Romero, Backward Trace Slicing for Conditional Rewrite Theories, in: Proc. of the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning LPAR-18, volume 7180 of *LNCS*, Springer, 2012, pp. 62–76.

[11] M. Alpuente, D. Ballis, F. Frechina, D. Romero, Julienne: A Trace Slicer for Conditional Rewrite Theorie, in: Proc. of the 118th Int'l Symposium on Formal Methods FM 2012, LNCS, Springer, 2012. To appear.

[12] TeReSe (Ed.), Term Rewriting Systems, Cambridge University Press, Cambridge, UK, 2003.

[13] J. Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, Theoretical Computer Science 96 (1992) 73–155.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, volume 4350 of *LNCS*, Springer-Verlag, 2007.

[15] J. Klop, Term Rewriting Systems, in: S. Abramsky, D. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, volume I, Oxford University Press, 1992, pp. 1–112.

[16] R. Bruni, J. Meseguer, Semantic Foundations for Generalized Rewrite Theories, Theoretical Computer Science 360 (2006) 386–414.

[17] F. Durán, J. Meseguer, A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories, in: Proc. of 8th International Workshop on Rewriting Logic and Its Applications (WRLA'10), number 6381 in LNCS, Springer, 2010, pp. 86–103.

[18] G. D. Plotkin, A Structural Approach to Operational Semantics, J. Log. Algebr. Program. (2004) 17–139.

[19] J. Meseguer, The Temporal Logic of Rewriting: A Gentle Introduction, in: Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of his 65th Birthday, volume 5065, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 354–382.

[20] M. Ducassé, Opium: An Extendable Trace Analyzer for Prolog, Journal of Logic Programming 39 (1999) 177–223.

[21] Y. A. Liu, S. D. Stoller, Eliminating Dead Code on Recursive Data, Science of Computer Programming 47 (2003) 221–242.

[22] B. Korel, J. Laski, Dynamic Program Slicing, Inf. Process. Lett. 29 (1988) 155–163.

[23] M. Alpuente, D. Ballis, F. Frechina, D. Romero, The JULIENNE web site, 2012. Available at: `http://safe-tools.dsic.upv.es/julienne/`.

[24] S. Escobar, C. Meadows, J. Meseguer, Maude-NPA: Cryptographic Protocol Analysis Modulo Equational properties, in: FOSAD 2007/2008/2009 Tutorial Lectures, volume 258(1) of *LNCS*, Springer, 2009, pp. 1–50.

[25] G. Lowe, Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR, in: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems, volume 1055 of *LNCS*, Springer, 1996, pp. 147–166.

[26] I. Bethke, J. W. Klop, R. de Vrijer, Descendants and origins in term rewriting, Inf. Comput. 159 (2000) 59–124.

[27] J. Field, F. Tip, Dynamic dependence in term rewriting systems and its application to program slicing, in: Proc. of the 6th Int'l Symposium on Programming Language Implementation and Logic Programming, PLILP '94, Springer-Verlag, London, UK, 1994, pp. 415–431.

[28] H. Agrawal, R. A. DeMillo, E. H. Spafford, Debugging with Dynamic Slicing and Backtracking, Softw., Pract. Exper. 23 (1993) 589–616.

[29] O. Chitil, C. Runciman, M. Wallace, Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs, in: Implementation of Functional Languages, 12th International Workshop, IFL 2000, volume 2011 of *LNCS*, Springer, 2000, pp. 176–193.

[30] T. Wang, A. Roychoudhury, Jslice: A Dynamic Slicing tool for Java Programs, 2008. Available at: `http://jslice.sourceforge.net/`.

[31] F. Tip, A Survey of Program Slicing Techniques, J. Prog. Lang. 3 (1995).

[32] A. Riesco, A. Verdejo, N. Martí-Oliet, Algebraic methodology and software technology - 13th international conference, amast 2010. revised selected papers, in: M. Johnson, D. Pavlovic (Eds.), AMAST, volume 6486 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 216–225.

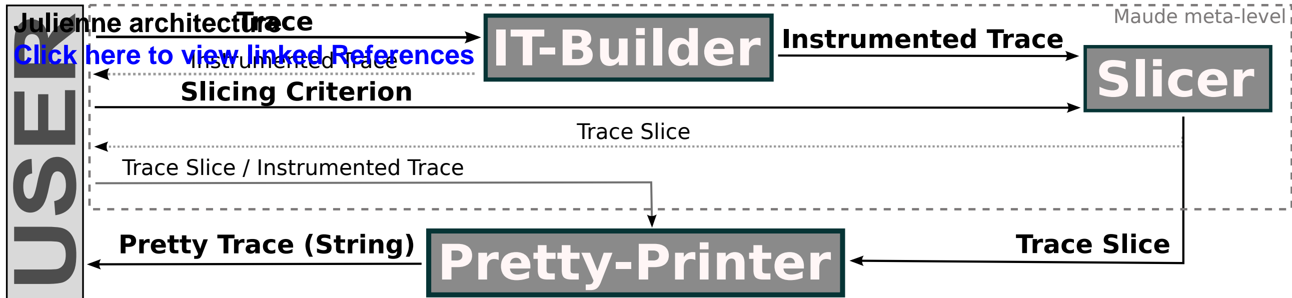
[33] E. Y. Shapiro, Algorithmic Program Diagnosis, in: Conference Record of Ninth Annual ACM Symposium on Principles of Programming Languages, POPL'82, pp. 299–308.

[34] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, volume 6418 of *Lecture Notes in Computer Science*, Springer, 2010.

[35] A. Farzan, F. Chen, J. Meseguer, G. Rosu, Formal analysis of Java programs in JavaFAN, in: CAV, pp. 501–505.

Julienne architecture

Click here to view linked References



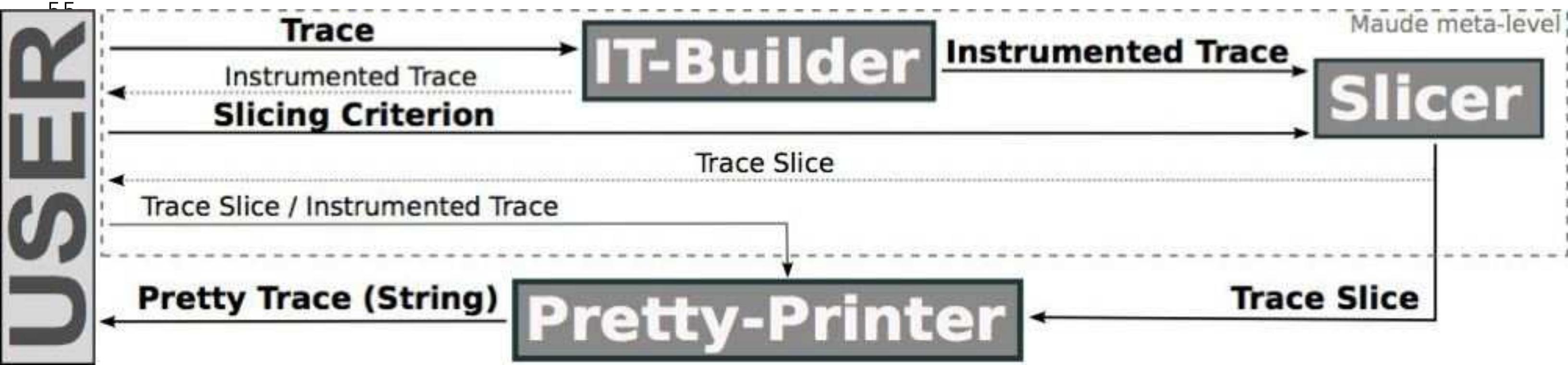
Step	RuleName	Execution trace	Sliced trace
1	'Start	ac(A,50) ; ac(B,20) ; ac(D,20) ; ac(C,20) ; credit(A,10) ; credit(D,40) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4)	ac(C,20) ; * ; * ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
2	unflattening	***deleted***	
3	credit	ac(B,20) ; ac(D,20) ; ac(C,20) ; credit(D,40) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(A,60)	ac(C,20) ; * ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
4/5	flattening / unflattening	***deleted***	
6	credit	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(D,60)	ac(C,20) ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
7/8	flattening / unflattening	***deleted***	
9	debitERR	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(D,55)	ac(C,20) ; * ; * ; * ; debit(C,50) ; transfer(*,C,4) ; transfer(*,C,15) ; *
10/11	flattening / unflattening	***deleted***	
12	debitERR	ac(A,60) ; ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(C,-30)	ac(C,-30) ; * ; * ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
13/14	flattening / unflattening	***deleted***	
15	transfer	ac(B,20) ; ac(C,-30) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(A,40) ; ac(D,75)	* ; ac(C,-30) ; * ; * ; transfer(*,C,4) ; transfer(*,C,15)
16/17	flattening / unflattening	***deleted***	
18	transfer	ac(B,20) ; ac(D,75) ; transfer(B,C,4) ; ac(A,25) ; ac(C,-15)	* ; * ; ac(C,-15) ; * ; transfer(*,C,4)
19/20	flattening / unflattening	***deleted***	
21	transfer	ac(A,25) ; ac(D,75) ; ac(B,16) ; ac(C,-11)	* ; * ; * ; ac(C,-11)
22	flattening	***deleted***	
		Condition:	TRUE
Total size:		1689	374
Reduction: 78%			

Step	Rule name	Execution trace	Sliced trace
1	'Start	minmax(4 ; 7 ; 0)	minmax(*23 ; *19 ; 0)
2	unflattening	minmax(4 ; 7 ; 0)	***deleted***
3	minmax2	PAIR(Min(4,1st(minmax(7 ; 0))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,1st(minmax(*19 ; 0))),Max(*1,*2))
4	minmax2	PAIR(Min(4,1st(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,1st(PAIR(Min(*8,1st(minmax(0))),*17))),Max(*1,*2))
5	1st	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,Min(*8,1st(minmax(0)))),Max(*1,*2))
6	minmax2	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))))	***deleted***
7	2nd	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,Max(7,2nd(minmax(0)))))	***deleted***
8	minmax1	PAIR(Min(4,Min(7,1st(PAIR(0,0)))),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,Min(*8,1st(PAIR(0,*11)))),Max(*1,*2))
9	1st	PAIR(Min(4,Min(7,0)),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,Min(*8,0)),Max(*1,*2))
10	Min1	PAIR(Min(4,0),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,0),Max(*1,*2))
11	Min1	PAIR(0,Max(4,Max(7,2nd(minmax(0)))))	PAIR(0,Max(*1,*2))
12	minmax1	PAIR(0,Max(4,Max(7,2nd(PAIR(0,0)))))	***deleted***
13	2nd	PAIR(0,Max(4,Max(7,0)))	***deleted***
14	Max1	PAIR(0,Max(4,7))	***deleted***
15	Max2	PAIR(0,7)	PAIR(0,*0)
		Condition:	*23 > 0 and *19 > 0
Total size:		679	312
Reduction: 54%			

Julienne architecture

Click here to view linked References

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54



Julienne output bank slice trace

Click here to view linked References

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

Step	RuleName	Execution trace	Sliced trace
1	'Start	ac(A,50) ; ac(B,20) ; ac(D,20) ; ac(C,20) ; credit(A,10) ; credit(D,40) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4)	ac(C,20) ; * ; * ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
2	unflattening	***deleted***	
3	credit	ac(B,20) ; ac(D,20) ; ac(C,20) ; credit(D,40) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(A,60)	ac(C,20) ; * ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
4/5	flattening / unflattening	***deleted***	
6	credit	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(D,5) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(D,60)	ac(C,20) ; * ; * ; * ; debit(C,50) ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
7/8	flattening / unflattening	***deleted***	
9	debitERR	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(D,55)	ac(C,20) ; * ; * ; * ; debit(C,50) ; transfer(*,C,4) ; transfer(*,C,15) ; *
10/11	flattening / unflattening	***deleted***	
12	debitERR	ac(A,60) ; ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(C,-30)	ac(C,-30) ; * ; * ; * ; transfer(*,C,4) ; transfer(*,C,15) ; *
13/14	flattening / unflattening	***deleted***	
15	transfer	ac(B,20) ; ac(C,-30) ; transfer(A,C,15) ; transfer(B,C,4) ; ac(A,40) ; ac(D,75)	* ; ac(C,-30) ; * ; * ; transfer(*,C,4) ; transfer(*,C,15)
16/17	flattening / unflattening	***deleted***	
18	transfer	ac(B,20) ; ac(D,75) ; transfer(B,C,4) ; ac(A,25) ; ac(C,-15)	* ; * ; ac(C,-15) ; * ; transfer(*,C,4)
19/20	flattening / unflattening	***deleted***	
21	transfer	ac(A,25) ; ac(D,75) ; ac(B,16) ; ac(C,-11)	* ; * ; * ; ac(C,-11)
22	flattening	***deleted***	
		Condition:	TRUE
Total size:		1689	374
Reduction: 78%			

Julienne output minmax slice trace
Click here to view linked References

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

Step	Rule name	Execution trace	Sliced trace
1	'Start	minmax(4 ; 7 ; 0)	minmax(*23 ; *19 ; 0)
2	unflattening	minmax(4 ; 7 ; 0)	***deleted***
3	minmax2	PAIR(Min(4,1st(minmax(7 ; 0))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,1st(minmax(*19 ; 0))),Max(*1,*2))
4	minmax2	PAIR(Min(4,1st(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,1st(PAIR(Min(*8,1st(minmax(0))),*17))),Max(*1,*2))
5	1st	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*6,Min(*8,1st(minmax(0)))),Max(*1,*2))
6	minmax2	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))))	***deleted***
7	2nd	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,Max(7,2nd(minmax(0)))))	***deleted***
8	minmax1	PAIR(Min(4,Min(7,1st(PAIR(0,0)))),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,Min(*8,1st(PAIR(0,*11)))),Max(*1,*2))
9	1st	PAIR(Min(4,Min(7,0)),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,Min(*8,0)),Max(*1,*2))
10	Min1	PAIR(Min(4,0),Max(4,Max(7,2nd(minmax(0)))))	PAIR(Min(*6,0),Max(*1,*2))
11	Min1	PAIR(0,Max(4,Max(7,2nd(minmax(0)))))	PAIR(0,Max(*1,*2))
12	minmax1	PAIR(0,Max(4,Max(7,2nd(PAIR(0,0)))))	***deleted***
13	2nd	PAIR(0,Max(4,Max(7,0)))	***deleted***
14	Max1	PAIR(0,Max(4,7))	***deleted***
15	Max2	PAIR(0,7)	PAIR(0,*0)
		Condition:	*23 > 0 and *19 > 0
Total size:		679	312
Reduction: 54%			