Backward Trace Slicing for Conditional Rewrite Theories*

Mará Alpuente[1], Demis Ballis[2], Francisco Frechina[1], and Daniel Romero[1]

[1] DSIC-ELP, Universitat Politècnica de València,
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
{alpuente,ffrechina,dromero}@dsic.upv.es
[2] DIMI, Università degli Studi di Udine,
Via delle Scienze 206, 33100 Udine, Italy
demis.ballis@uniud.it

Abstract. In this paper, we present a trace slicing technique for rewriting logic that is suitable for analyzing complex, textually-large system computations in rewrite theories that may contain conditional equations and/or rules. Given a conditional execution trace $\mathcal{T}$ and a slicing criterion for the trace (i.e., a set of positions that we want to observe in the final state of the trace), we traverse $\mathcal{T}$ from back to front, and at each rewrite step, we incrementally compute the origins of the observed positions, which is done by inductively processing the conditions of the applied equations and rules. During the traversal, we also carry a boolean compatibility condition that is needed for the executability of the processed rewrite steps. At the end of the traversal, the trace slice is obtained by filtering out the irrelevant data that do not contribute to the criterion of interest.

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 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 [10].

Rewriting Logic (RWL) is a very general *logical* and *semantic framework*, which is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [7] and Web systems [2,5]). RWL is efficiently implemented in the high-performance system Maude [12]. Roughly speaking, a *(conditional) rewriting logic theory* [16] seamlessly combines a *(conditional) term*

* 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, and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004. Also, D. Romero is supported by FPI-MEC grant BES-2008-004860 and F. Frechina is supported by FPU-ME grant AP2010-5681.

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 axioms.

In recent years, the debugging and optimization techniques based on RWL have received growing attention. However, to the best of our knowledge, the only trace slicing technique that gives support to the analysis of RWL computations is [3]. Given an execution trace $\mathcal{T}$, [3] generates a trace slice of $\mathcal{T}$ w.r.t. a set of symbols of interest (target symbols) that appear in a given state of $\mathcal{T}$. The technique relies on a suitable mechanism of backward tracing that computes the reverse dependence among the symbols involved in an execution step by using a procedure that dynamically labels the calls (terms) involved in the steps.

Unfortunately, the technique in [3] 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, which is what has motivated our work.

Example 1. Consider the Maude specification of the function `_mod_` in Figure 1, which computes the reminder of the division of two natural numbers, and the associated execution trace 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 [3] computes the trace slice 4 `mod` $\bullet$ $\to$ 4, whereas the correct trace slice is 4 `mod` 5 $\to$ 4 since both arguments of `mod` are required to prove the rewrite step that introduces the symbol 4 in the final state.

```
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
```

Fig. 1. The `_mod_` operator

Contributions. We present the first conditional trace slicing technique for RWL computations. Our technique is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates conditional RWL computations such as those delivered as counterexample traces by the Maude model-checker [6]. The backward conditional slicing algorithm in this paper cannot be considered to be a natural extension of the unconditional slicing method of [3], but greatly simplifies [3] by replacing the involved and costly dynamic labeling procedure, based on [8], with a simple mechanism for substitution refinement that allows control and data dependencies to be propagated between consecutive rewrite steps. Moreover, the conditional slicing algorithm copes with three different types of conditions that occur in Maude theories: equational conditions, matching conditions, and rewrite expressions. Our formulation takes into account the precise way in which Maude mechanizes the conditional rewriting process and revisits all those rewrite steps backwards in an instrumented, fine-grained way where each small step corresponds to the application of an equation (conditional equation or equational axiom) or rule. This allows us to slice the input execution trace with regard to the set of symbols of interest (target symbols) by tracing back the target symbols along the execution trace so that all data that are not antecedents of the observed symbols are simply discarded.

Related Work. Tracing techniques have been extensively used in functional debugging [11]. For instance, Hat [11] is an interactive debugging system that enables a computation to be explored backwards, starting from the program output or an error message (with which the computation aborted). Backward tracing in Hat is carried out by navigating a redex trail (i.e., a graph-like data structure that records dependencies among function calls), whereas our tracing technique does not require handling any supplementary data structure.

There exist very few approaches that address the problem of tracing rewrite sequences in term rewrite systems [3,8,14,18], and all of them apply to unconditional systems. The techniques in [3,8,18] rely on a labeling relation on symbols that allows data content to be traced back within the computation; this is achieved in [14] by formalizing a notion of dynamic dependence among symbols by means of contexts. In [8,18], non-left linear and collapsing rules are not considered or are dealt using ad-hoc strategies, while our approach requires no special treatment of such rules. Furthermore, only [3] describes a tracing methodology for rewrite theories with rules, equations, sorts, and algebraic axioms.

In this paper, we propose a more general slicing technique for conditional rewrite theories that generalizes and simplifies the formal development in [3] by getting rid of the complex dynamic labeling algorithm that was needed to trace back the origins of the symbols of interest. 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.

To debug Maude programs, Maude has a tracing facility that allows the execution sequence to be traced, and is very customizable: it provides some control over conditions and allows the user to select the statements being applied at each step. A main difference with the trace slicing technique of ours is that the tracer of Maude allows the trace size to be reduced by manually focusing on statements, while slicing is automatic and focuses on terms. Moreover, since each small rewrite step that is obtained by applying a single conditional equation, equational axiom or rule is shown in the trace, the user can easily miss the general view, and when the user detects an incorrect intermediate result, it is difficult to know where the incorrect inference started. In this regard, the trace slices computed by our technique can be very helpful in debugging, since they only consist of the information that is strictly needed to deliver a critical part of the result (see discussion in [1]).

Plan of the Paper. 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. Finally, Section 5 reports on a prototypical implementation of the proposed slicing technique and its experimental evaluation.

2 Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [18] and Rewriting Logic [16]. Some familiarity with the Maude language [12] is also required.

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [12]. 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 $p, t \in \tau(\Sigma, \mathcal{V})$.

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

[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.

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 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 analogous 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.

3 Conditional Rewriting Modulo Equational Theories

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. The following Maude rewrite theory defines a simple banking system. It includes three conditional rules: `credit`, `debit`, and `transfer`.

```
mod BANK is inc INT .
  sorts Account Msg State Id .
  subsorts  Account Msg < State .
  var Id Id1 Id2 : Id .
  var bal bal1 bal2 newBal newBal1 newBal2 M : Nat .
  op empty-state : -> State .
  op _;_ : State State -> State [assoc comm id: empty-state] .
  op <_|_> : Id Nat -> Account [ctor] .
  ops credit debit : Id Nat -> Msg [ctor] .
  op  transfer : Id Id Nat -> Msg  [ctor] .
  crl [credit] : <Id|bal>;credit(Id,M) => <Id|newBal> if newBal := bal + M .
  crl [debit] : <Id|bal>;debit(Id,M) => <Id|newBal> if bal >= M /\ newBal := bal - M .
  crl [transfer] : <Id1|bal1>;<Id2|bal2>;transfer(Id1,Id2,M) => <Id1|newBal1>;<Id2|newBal2>
   if <Id1|bal1>;debit(Id1,M) => <Id1|newBal1> /\ <Id2|bal2>;credit(Id2,M) => <Id2|newBal2> .
endm
```

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

The rule `credit` contains a matching condition `newBal := bal + M`. The rule `debit` contains an equational condition `bal >= M` and a matching condition `newBal := bal - M`. Finally, the rule `transfer` has a rule condition that contains two rewrite expressions `<Id1|bal1>;debit(Id1,M) => <Id1|newBal1>` and `<Id2|bal2>;credit(Id2,M) => <Id2|newBal2>`.

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 $=_E$ [9], that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is in general undecidable, since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The conditional slicing technique formalized in this work is formulated by considering the precise way in which Maude proves the conditional rewriting steps (see Section 5.2 in [12]). Actually, 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 [12].

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 $e\sigma{\downarrow_\Delta}{=_B}$ *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 $t\sigma \rightarrow^*_{R\cup\Delta,B} u$, 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.

[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.

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 [13]: (i) reduce t w.r.t. $\rightarrow_{\Delta,B}$ until the canonical form $t\downarrow_{\Delta}$ is reached; (ii) rewrite $t\downarrow_{\Delta}$ w.r.t. $\rightarrow_{R,B}$.

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

$$s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_{\Delta}} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_{\Delta}} \ldots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ 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 [3]. This way, any given instrumented execution trace consists of a sequence of (standard) rewrites using the conditional equations ($\rightarrow_{\Delta}$), conditional rules ($\rightarrow_{R}$), and axioms ($\rightarrow_{B}$).

Example 3. Consider the rewrite theory in Example 2 together with the following execution trace $\mathcal{T}$: `credit(A,2+3);<A|10>` $\rightarrow_{\Delta,B}$ `credit(A,5);<A|10>` $\rightarrow_{R,B}$ `<A|15>` Thus, the corresponding instrumented execution trace is given by expanding the commutative "step" applied to the term `credit(A,2+3);<A|10>` using the implicit rule (`X;Y` $\rightarrow$ `Y;X`) in B that models the commutativity axiom for the (juxtaposition) operator _;_.

`credit(A,2+3);<A|10>` $\rightarrow_{\Delta}$`credit(A,5);<A|10>` $\rightarrow_{B}$`<A|10>;credit(A,5)` $\rightarrow_{R}$`<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 [3].

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 $f(b, f(f(b,a),c)) \rightarrow^*_{\mathit{flat}_B} f(a,b,b,c) \rightarrow^*_{\mathit{unflat}_B} f(f(b,c), f(a,b))$, where the first one corresponds to a *flattening* transformation sequence that obtains the AC canonical form, while the second one 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 $\rightarrow_{\Delta}$, $\rightarrow_{R}$, $\rightarrow_{B}$ by $\rightarrow$. Also, by $\rightarrow^*$ (resp. $\rightarrow^+$), we denote the transitive and reflexive (resp. transitive) closure of the relation $\rightarrow_{\Delta} \cup \rightarrow_{R} \cup \rightarrow_{B}$.

4 Backward Conditional Slicing

In this section, we formulate our backward 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 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 $\mathit{fresh}^{\bullet}$[4]. More formally, a term slice is defined as follows.

Definition 1 (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:*

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

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

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. 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 1, we get the term slice $t^{\bullet} = \mathit{slice}(t, P) = d(f(g(\bullet_1, \bullet_2), y), \bullet_3)$ 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 2 (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.*

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

4.2 Backward Slicing for Execution Traces

Consider an execution trace $\mathcal{T} : s_0 \rightarrow^* s_n$. 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 3. *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} s_1 \overset{r_2,\sigma_2,w_2}{\rightarrow} \ldots \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ 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 $[s_0^\bullet \rightarrow s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B^\bullet]$, 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 $s'_0 \rightarrow s'_1 \rightarrow \ldots \rightarrow s_n$ in $\mathcal{R}$ such that*
   i) $s'_i \rightarrow s'_{i+1}$ is either the rewrite step $s'_i \overset{r_{i+1},\sigma'_{i+1},w_{i+1}}{\rightarrow} s'_{i+1}$ 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 3 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 6. Consider the Maude specification of Example 2 together with the following execution trace $\mathcal{T}$: `(<a|30>;debit(a,5));credit(a,3)` $\overset{\texttt{debit}}{\rightarrow}$ `<a|25>; credit(a,3)` $\overset{\texttt{credit}}{\rightarrow}$ `<a|28>`. Let `<a|`$\bullet_1$`>` be a term slice of `<a|28>` generated with the slicing criterion $\{1\}$ —i.e., `<a|`$\bullet_1$`>`$=slice($`<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

`(<a|`$\bullet_8$`>;debit(a,`$\bullet_9$`));credit(a|`$\bullet_4$`)`$\overset{\texttt{debit}}{\rightarrow}$`<a|`$\bullet_3$`>;credit(a,`$\bullet_4$`)`$\overset{\texttt{credit}}{\rightarrow}$`<a|`$\bullet_1$`>`

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 `<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 $(Conf,\ \bullet\!\!\rightarrow)$ [17] 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. *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 $(\mathit{Conf},\ \bullet\!\!\rightarrow)$ is defined as follows.

Definition 5. *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* $W^\bullet$ *be a term slice of* W*. Then, the transition relation* $\bullet\!\!\rightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ *is the smallest relation that satisfies the following rule:*

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

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 $B_W^\bullet$.

The initial configuration $\langle s_0 \rightarrow^* s_n,\ \mathit{slice}(s_n, \mathcal{O}_{s_n}),\ \mathit{true} \rangle$ 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 $s_0 \rightarrow^* s_n$. 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 6 (Backward trace slicing algorithm). *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory, and let* $\mathcal{T} : s_0 \rightarrow^* s_n$ *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:*

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

iff there exists a transition sequence in $(\mathit{Conf},\ \bullet\!\!\rightarrow)$

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

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 2, takes as input three parameters, namely, a rewrite step $\mu : s \overset{r,\sigma,w}{\to} t$ (with $r = \lambda \to \rho \text{ } if \text{ } C$[5]), a term slice $t^\bullet$ of t, and a 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.

```
function slice-step($s \overset{r,\sigma,w}{\to} t,\ t^\bullet,\ B^\bullet_{prev}$)
 1. if $w \notin \mathcal{MP}os(t^\bullet)$
 2.   then
 3.     $s^\bullet = t^\bullet$
 4.     $B^\bullet = B^\bullet_{prev}$
 5.   else
 6.     $\theta = \{x/fresh^\bullet \mid x \in Var(r)\}$
 7.     $\rho^\bullet = slice(\rho, \mathcal{MP}os(t^\bullet_{|w}))$
 8.     $\psi_\rho = \langle\!| \theta, match_{\rho^\bullet\theta}(t^\bullet_{|w}) |\!\rangle$
 9.     for $i = n$ downto 1 do
10.       $(\psi_i,\ B^\bullet_i) = process\text{-}condition(c_i,\ \sigma,$
                                  $\langle\!| \psi_\rho, \psi_n ... \psi_{i+1} |\!\rangle)$
11.     od
12.     $s^\bullet = t^\bullet[\lambda \langle\!| \psi_\rho, \psi_n ... \psi_1 |\!\rangle]_w$
13.     $B^\bullet = (B^\bullet_{prev} \wedge B^\bullet_n ... \wedge B^\bullet_1)(\psi_1 \psi_2 \ldots \psi_n)$
14. fi
15. return $(s^\bullet,\ B^\bullet)$
```

Fig. 2. Backward step slicing function

Definition 7 (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:* $\langle\!| \sigma_1, \sigma_2 |\!\rangle = \sigma_{\restriction Dom(\sigma_1)}$*, where*

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

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

Example 7. 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 8. Consider the Maude specification of Example 2 and the following rewrite step μ: `(<a|30>;debit(a,5));credit(a,3)` $\overset{\texttt{debit}}{\to}$ `<a|25>;credit(a,3)`. Let $\bullet_1$; `credit(a,3)` be a term slice of `<a|25>;credit(a,3)`. Since the rewrite step μ occurs at position $1 \notin \mathcal{MP}os(\bullet_1;$ `credit(a,3)`$)$, the term `<a|25>` introduced by μ in `<a|25>;credit(a,3)` is completely ignored in $\bullet_1$; `credit(a,3)`. Hence, the

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

computed term slice for `(<a|30>;debit(a,5));credit(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.

```
function process-condition($c$, $\sigma$, $\theta$)
  case $c$ of
  $(p := t) \vee (t \Rightarrow p)$ :
     if ($t\sigma = p\sigma$)
        then return ({}, true) fi
     $Q = \mathcal{MP}os(p\theta)$
     $[t^\bullet \to^* p^\bullet,\ B^\bullet] =$
           backward-slicing($t\sigma \to^* p\sigma$, $Q$)
     $t^{\bullet\prime} = slice(t, \mathcal{MP}os(t^\bullet))$
     $\psi = match_{t^{\bullet\prime}\theta}(t^\bullet)$
  $e$ :
     $\psi = \{\ \}$
     $B^\bullet = e\theta$
  end case
  return ($\psi$, $B^\bullet$)
```

Fig. 3. Condition processing function

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 9. Consider the rewrite theory in Example 2 together with the following rewrite step $\mu_{\texttt{debit}}$: `<a|30>;debit(a,5)` $\overset{\texttt{debit}}{\to}$ `<a|25>` that involves the application of the `debit` rule whose right-hand side is $\rho_{\texttt{debit}}$ =`<Id|newBal>`. Let $t^\bullet$ =`<a|`$\bullet_1$`>` be a term slice of `<a|25>`. Then, the initially ascertained substitution for μ is θ=$\{\texttt{Id}/\bullet_2, \texttt{bal}/\bullet_3, \texttt{M}/\bullet_4, \texttt{newBal}/\bullet_5\}$, and $match_{\rho_{\texttt{debit}}\theta}(t^\bullet) = match_{\texttt{<}\bullet_2\texttt{|}\bullet_5\texttt{>}}(\texttt{<a|}\bullet_1\texttt{>}) = \{\bullet_2/\texttt{a},\ \bullet_5/\bullet_1\}$. Thus, the substitution $\psi_{\rho_{\texttt{debit}}} = \langle\!| \theta, \psi_{\rho_{\texttt{debit}}} |\!\rangle = \{\texttt{Id}/\texttt{a},\ \texttt{bal}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{newBal}/\bullet_1\}$. That is, $\psi_{\rho_{\texttt{debit}}}$ refines θ by replacing the uninformed binding $\texttt{Id}/\bullet_2$, with $\texttt{Id}/\texttt{a}$.

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 3 that generates a pair $(\psi_i,\ B^\bullet_i)$ such that ψ_i is used to further refine the partially ascertained substitution $\langle\!| \psi_\rho, \psi_n, \ldots, \psi_{i+1} |\!\rangle$ 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 $\langle\!| \psi_\rho, \psi_n, \ldots, \psi_1 |\!\rangle$, 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 $\psi_1 \dots \psi_n$. It is worth noting that *process-condition* handles rewrite expressions, equational conditions, and matching conditions differently. More specifically, the pair $(\psi_i,\ B_i)$ 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, $[m^\bullet \to^* p^\bullet,\ B^\bullet] = \textit{backward-slicing}(m\sigma \to^* p\sigma,\ \mathcal{MP}os(p\theta))$. 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 $\psi = \textit{match}_{m\theta}(m^\bullet)$, which is the substitution needed to refine θ, and then the pair $(\psi,\ B^\bullet)$ is returned.

  Example 10. Consider the the rewrite step $\mu_{\texttt{debit}}$ of Example 9 together with the refined substitution $\theta = \{\texttt{Id/a},\ \texttt{bal/}\bullet_3,\ \texttt{M/}\bullet_4,\ \texttt{newBal/}\bullet_1\}$. We process the condition `newBal := bal - M` of `debit` by considering an internal execution trace $\mathcal{T}_{int} = \texttt{30} - \texttt{5} \to \texttt{25}$ [6]. By invoking the function *backward-slicing* the trace slice result is $[\bullet_6 \to \bullet_6,\ \textit{true}]$. The final outcome is given by $\textit{match}_{\bullet_7 - \bullet_8}(\bullet_6)$, that is $\textit{fail}$. 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 $\mu : s \overset{r,\sigma,w}{\to} t$, 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.

[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.

Example 11. Consider the refined substitution given in Example 10 $\theta = \{\texttt{Id/a},\ \texttt{bal/}\bullet_3,\ \texttt{M/}\bullet_4,\ \texttt{newBal/}\bullet_1\}$ together with the rewrite step $\mu_{\texttt{debit}}$ of Example 9 that involves the application of the `debit` rule. After processing the condition `bal >= M` of `debit`, we deliver $B^\bullet = (\bullet_3 >= \bullet_4)$.

Soundness of our conditional slicing technique is established by the following theorem. The proof can be found in [4].

Theorem 1 (soundness). *Let $\mathcal{R}$ be a rewrite theory. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} \ldots \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ 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 $\mathcal{T}$.*

5 Implementation and Experimental Evaluation

The conditional slicing methodology presented so far has been implemented in a prototype tool that is written in Maude and publicly available at `http://users.dsic.upv.es/grupos/elp/soft.html`.

The prototype takes in input a slicing criterion and a Maude execution trace, which is a term of sort `Trace` (generated by means of the the Maude metalevel operator `metaSearchPath`), and delivers the corresponding trace slice.

We have tested our prototype on rather large execution traces, such as the counterexamples generated by the model checker for Web applications Web-TLR [2]. In our experiments, we have considered a Webmail application together with four LTLR properties that have been refuted by Web-TLR. For each refuted property, Web-TLR has produced the corresponding counterexample in the form of a huge, textual execution trace $\mathcal{T}_i$, $i = 1, \ldots, 4$, in the range $10 - 100Kb$ that has been used to feed our slicer.

Table 1. Backward trace slicing benchmarks

Example trace	**Original trace size**	**Slicing criterion**	**Sliced trace size**	**%** ***reduction***
`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 1 shows the size of the original counterexample trace and that of the computed trace slice, both measured as the length of the corresponding string, w.r.t. two slicing criteria, that are detailed in the tool website. The considered criteria allow one to monitor the messages exchanged by a specific Web browser and the Webmail server, as well as to isolate the changes on the data structures of the two interacting entities. The *%reduction* column in Table 1 refers to the percentage of reduction achieved. The results we have obtained are very encouraging, and show an impressive reduction rate (up to $\sim 95\%$) in reasonable time (max. 0.9s on a Linux box equipped with an Intel Core 2 Duo 2.26GHz and 4Gb of RAM memory). Actually, sometimes the trace slices are small enough to be easily inspected by the users, who can restrict their attention to the part of the computation that they want to observe.

References

1. Alpuente, M., Ballis, D., Espert, J., Frechina, F., Romero, D.: Debugging of Web Applications with WEB-TLR. In: 7th Int'l Workshop on Automated Specification and Verification of Web Systems WWV 2011. EPTCS, vol. 61, pp. 66–80 (2011)
2. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Model-Checking Web Applications with Web-TLR. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 341–346. Springer, Heidelberg (2010)
3. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward Trace Slicing for Rewriting Logic Theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 34–48. Springer, Heidelberg (2011)
4. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Trace Slicing of Conditional Rewrite Theories. Tech. rep., Universidad Politécnica de Valencia (2012)
5. Alpuente, M., Ballis, D., Romero, D.: Specification and Verification of Web Applications in Rewriting Logic. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 790–805. Springer, Heidelberg (2009)
6. Bae, K., Meseguer, J.: A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. In: 9th Int'l Workshop on Rule-Based Programming RULE 2008. ENTCS. Elsevier (2008)
7. Baggi, M., Ballis, D., Falaschi, M.: Quantitative Pathway Logic for Computational Biology. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 68–82. Springer, Heidelberg (2009)
8. Bethke, I., Klop, J.W., de Vrijer, R.: Descendants and origins in term rewriting. Inf. Comput. 159(1-2), 59–124 (2000)
9. Bruni, R., Meseguer, J.: Semantic Foundations for Generalized Rewrite Theories. Theoretical Computer Science 360(1–3), 386–414 (2006)
10. Chen, F., Roşu, G.: Parametric Trace Slicing and Monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009)
11. Chitil, O., Runciman, C., Wallace, M.: Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 176–193. Springer, Heidelberg (2001)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talco, C.: Maude Manual (Version 2.6). Tech. rep., SRI Int'l Computer Science Laboratory (2011), `http://maude.cs.uiuc.edu/maude2-manual/`
13. Durán, F., Meseguer, J.: A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 86–103. Springer, Heidelberg (2010)
14. Field, J., Tip, F.: Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. In: Penjam, J. (ed.) PLILP 1994. LNCS, vol. 844, pp. 415–431. Springer, Heidelberg (1994)
15. Klop, J.: Term Rewriting Systems. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, vol. I, pp. 1–112. Oxford University Press (1992)
16. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
17. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program., 17–139 (2004)
18. TeReSe (ed.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)