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 do not impact on 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 that we execute by using *i*Julienne, a trace slicer that implements our conditional slicing technique and is endowed with a trace querying mechanism that increases flexibility and reduction power.

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

[☆]This work has been partially supported by the EU (FEDER) and the Spanish MEC project ref. TIN2010-21062-C02-02, and by Generalitat Valenciana ref. PROMETEO2011/052. This work was carried out during the tenure of D. Ballis' ERCIM "Alain Bensoussan" Postdoctoral Fellowship. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 246016. F. Frechina was supported by FPU-ME grant AP2010-5681 and D. Romero by FPI-MEC grant BES-2008-004860.

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)

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 [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 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 in Maude (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, {}>` $\rightarrow$ `<skip, {}>` $\rightarrow$ {}, 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 some incorrect output. In conventional programming environments, for such an analysis, the programmer must repeatedly determine which statements in the code impact 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 dependency 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 locating 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] and a totally redesigned implementation of our conditional trace slicing technique [11] that we called *i*Julienne [12].

Our proposal for conditional trace slicing is aimed at endowing the RWL framework with a new instrument that can be used to improve 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 a novel 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 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 such rewrite steps are revisited backwards in an instrumented, fine-grained way.
(c) Unlike previous backward tracing approaches [9, 13], which are based on a costly, dynamic labeling procedure, here a less expensive, incremental technique of matching refinement is used that allows the relevant data to be traced back.

2. We have developed the *i*Julienne system, which is the first slicing-based, incremental trace analysis tool that can be used to analyze execution traces of RWL-based programs and tools. *i*Julienne supersedes and greatly improves the preliminary system presented in [11]. The original algorithm was developed under the assumption that the user examines and slices the entire trace w.r.t. a static slicing criterion, in a non-incremental way. In contrast, the slicing criterion in *i*Julienne can be dynamically changed, which provides the user with a stepwise focus on the information that she wants to observe at any rewrite step. The main features provided by the trace analyzer *i*Julienne are listed below.
   (a) *i*Julienne is equipped with an *incremental* backward trace slicing algorithm that supports incremental refinements of the trace slice and achieves huge reductions in the size of the trace. Starting from a Maude execution trace $\mathcal{T}$, a slicing criterion $\mathcal{S}$ can be attached to any given state of the trace and the computed trace slice $\mathcal{T}^\bullet$ can be repeatedly refined by applying backward trace slicing w.r.t. increasingly restrictive versions of $\mathcal{S}$.
   (b) The system supports a cogent form of *dynamic program slicing* [14] as follows. Given a Maude program $\mathcal{M}$ and a trace slice $\mathcal{T}^\bullet$ for $\mathcal{M}$, *i*Julienne is able to infer a fragment of $\mathcal{M}$ (i.e., the *program slice*) that is needed to reproduce $\mathcal{T}^\bullet$. This is done by uncovering statement dependence among computationally related parts of the program via backward trace slicing. This feature greatly facilitates the debugging of faulty Maude programs, since the user can generate a sequence of increasingly smaller program slices that gradually shrinks the area that contains the buggy piece of code.
   (c) *i*Julienne is endowed with a powerful and intuitive Web user interface that allows the slicing criteria to be easily defined by either highlighting the chosen target symbols or by applying a user-defined filtering pattern. A browsing facility is also provided that enables forward and backward navigation through the trace (and the trace slice) and allows the user to examine all the information that is involved within each state transition (and its corresponding sliced

counterpart) for debugging and comprehension purposes. The user interface can be tuned to provide distinct abstract views of the trace that aim at supporting different program comprehension levels. For instance, this includes hiding or displaying the auxiliary transformations that are used by Maude to handle associative and commutative operators.

3. To give the reader a better feeling of the generality and wide application range of our conditional slicing approach, we describe some applications of *i*JULIENNE to debugging, analysis and improvement of complex systems (e.g., biological systems, 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, and its formal correctness is stated. Section 5 describes *i*JULIENNE, together with some experiments that we have 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. The correctness of the proposed backward trace slicing methodology is formally proven in Appendix A.

2. Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [13] and Rewriting Logic [15]. Some familiarity with the Maude language [7] is also required. Maude [16] is a rewriting logic [15] 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]. The connected components of $(S, <)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_<$ containing $<$. 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. A simple syntactic condition on $(\Sigma, S, <)$ called preregularity [7] ensures that each (well-formed) term t has always a least-sort possible among all sorts in S, which is denoted $ls(t)$. 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 arises.

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 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 in t.

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 $Dom(\sigma) = \{X_1, \ldots, X_n\}$. The identity substitution is denoted by $\{\}$. The term $t\sigma$ is the result of applying σ to term t. For any substitution σ and set of variables V, $\sigma_{\restriction V}$ denotes the substitution obtained from σ by restricting its domain to V, (i.e., $\sigma_{\restriction V}(X) = X\sigma$ if $X \in V$, otherwise $\sigma_{\restriction V}(X) = X$). Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. The term t is an *instance* of the term s, iff there exists a matcher σ of t in s. By $match_s(t)$, we denote the function that returns a matcher of t in s if such a matcher exists.

We consider three different kinds of condition 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 $p, t \in \tau(\Sigma, \mathcal{V})$. Matching conditions an rewrite expressions are useful for performing a search through a structure without any need to explicitly define a function that does it. This is because substitutions for matching p against $t\sigma$ need not to be unique since some operators may be matched modulo equational attributes. For instance, considering that list concatenation obeys associativity with identity element `nil`, we can define two Maude equations to determine whether an element occurs in a list, as follows:

```
ceq member(e,lst) = true if (l1,e,l2) := lst .
eq member(e,lst) = false [owise] .
```

Unlike matching conditions that can only use equations to evaluate the input term t, rewrite expressions can apply both equations and rewrite rules for the evaluation.

2.2. Program Equations and Rules

A *conditional* equation is an expression of the form $\lambda = \rho$ *if* C, where $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$ (and $ls(\lambda) \equiv_< ls(\rho)$), 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)

[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 conjunction 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

$\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{i=1}^{n} \mathcal{V}ar(c_i)$, 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 $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$ (and $ls(\lambda) \equiv_< ls(\rho)$), 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 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 of an admissible rule/equation will become instantiated whenever the rule 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].

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

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 $\texttt{e}_1;\texttt{e}_2;\ldots...;\texttt{e}_n$. Each element $\texttt{e}_i$ is either (i) a bank account `ac(Id,b)`, where `ac` is a constructor symbol (denoted by the Maude `ctor` attribute), `Id` is 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 [17] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [18], 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, $t \stackrel{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) iff $\lambda\sigma =_B t_{|w}$, $t' = t[\rho\sigma]_w$, and C *evaluates*

to true w.r.t σ. When no confusion arises, we simply write $t \rightarrow_{R,B} t'$ (resp. $t \rightarrow_{\Delta,B} t'$) instead of $t \stackrel{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$).

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.

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 [19]: (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

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

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 [9]. 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 3.1

Consider the rewrite theory in Example 2.1 together with the following execution trace $\mathcal{T}$: `credit(A,2+3);ac(A,10)` $\rightarrow_{\Delta,B}$ `credit(A,5);ac(A,10)` $\rightarrow_{R,B}$ `ac(A,15)`. 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` $\rightarrow$ `Y;X` in B that models the commutativity axiom for the (juxtaposition) operator _;_.

`credit(A,2+3);ac(A,10)` $\rightarrow_\Delta$`credit(A,5);ac(A,10)` $\rightarrow_B$`ac(A,10);credit(A,5)` $\rightarrow_R$`ac(A,15)`

[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 $x \in Dom(\sigma)$, 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

Also, typically hidden inside the B-matching algorithms, some transformations allow terms that contain operators that obey associative and commutative (AC) 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 $f(b, f(f(b, a), c)) \rightarrow^*_{flat_B} f(a, b, b, c) \rightarrow^*_{unflat_B} f(f(b, c), f(a, b))$, where the first transformation corresponds to a *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 $\rightarrow_\Delta$, $\rightarrow_R$, $\rightarrow_B$ by $\rightarrow$. Also, we denote the transitive and reflexive (resp. transitive) closure of the relation $\rightarrow_\Delta \cup \rightarrow_R \cup \rightarrow_B$ by $\rightarrow^*$ (resp. $\rightarrow^+$).

4. Backward Conditional Trace 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 $\mathcal{T} : \texttt{4 mod 5} \rightarrow \texttt{4}$. 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 $\mathcal{T}^\bullet : \texttt{4 mod} \bullet \rightarrow \texttt{4}$, 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` $\not\rightarrow$ `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 $[4\ \texttt{mod}\ \bullet \to 4, \bullet > 4]$.

In this section, we formulate our slicing algorithm for conditional 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 of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$ More formally, let $\mathcal{V}^\bullet$ be the extension of the S-sorted family $\mathcal{V}$ that augments $\mathcal{V}$ with $\bullet$-variables in the obvious way. A term $t^\bullet$ is a *term slice* of the term t iff t is an instance of $t^\bullet$ and $t^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$. By $\mathcal{V}ar^\bullet(exp)$ we define the set of all $\bullet$-variables that occur in the expression exp.

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 all $i = 0, 1, \ldots$. 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. Let $\mathcal{M}^\bullet$ be a set of $\bullet$-variables. By $\overline{\mathcal{MP}os}(\mathcal{M}^\bullet, t^\bullet)$, we define the *extension* of $\mathcal{MP}os(t^\bullet)$ w.r.t. $\mathcal{M}^\bullet$ that we define as follows:

$$\overline{\mathcal{MP}os}(\mathcal{M}^\bullet, t^\bullet) = \mathcal{MP}os(t^\bullet) \cup \{p \in \mathcal{P}os(t^\bullet) \mid t^\bullet_{|p} \in \mathcal{M}^\bullet\}$$

The next auxiliary definition formalizes the function $slice(t, P)$ that allows a term slice of t to be constructed w.r.t. a set of position P of t. The function *slice* relies on the function $fresh^\bullet$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort, which is distinct from any previously generated variable $\bullet_j$.

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

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

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

and $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$ is the prefix closure *of P.*

Roughly speaking, $slice(t, P)$ yields a term slice of t w.r.t. a set of positions P that includes all symbols of t which 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 freshly generated $\bullet$-variable.

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 $\mathcal{MP}os(t^\bullet) = \{\Lambda, 1, 1.1, 1.2\}$. Let $\mathcal{M}^\bullet = \{\bullet_1, \bullet_3\}$. Then, $\overline{\mathcal{MP}os}(\mathcal{M}^\bullet, t^\bullet) = \{\Lambda, 1, 1.1, 1.2, 1.1.1, 2\}$.

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.*

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 $\mathcal{T} : s_0 \rightarrow^* s_n$ and a term slice $s_n^\bullet$ of s_n, generates the sliced counterpart $\mathcal{T}^\bullet : s_0^\bullet \rightarrow^* s_n^\bullet$ 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 correctness of the generated trace slice.

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* for $\mathcal{T}$ —i.e., a user-defined term slice $s_n^\bullet$ of s_n that only includes those symbols of s_n that we want to observe.

A trace slice is formally defined as follows.

Definition 4.6 (trace slice) *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 $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. A trace slice of $\mathcal{T}$ w.r.t. $s_n^\bullet$ is a pair $[s_0^\bullet \rightarrow s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B^\bullet]$, where $s_i^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$, with $i = 0, \ldots, n-1$, and $B^\bullet$ is a Boolean condition;*

The following definitions provide two notions of correctness for a trace slice that will be used to prove the correction of the backward trace slicing algorithm described in this section.

Definition 4.7 *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 $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. A trace slice $[s_0^\bullet \rightarrow s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B^\bullet]$ of $\mathcal{T}$ w.r.t. $s_n^\bullet$ is* weakly correct *iff there exists a term s'_0 with $s_0^\bullet \propto^{B^\bullet} s'_0$, and 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) $s_i^\bullet$ is a term slice of s'_i, for all $i = 0, \ldots, n$.

Definition 4.7 provides an existential condition on trace slices. Roughly speaking, it ensures that for *some* concrete trace $\mathcal{T}'$, whose first state s'_0 is a concretization of $s_0^\bullet$ which is compatible with $B^\bullet$, the rules involved in the sliced steps of $\mathcal{T}^\bullet$ can be applied again, at the same positions, in $\mathcal{T}'$ and each $s_i^\bullet$ is a term slice of s'_i.

Definition 4.8 *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 $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. A* trace slice *$[s_0^\bullet \rightarrow s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B^\bullet]$ of $\mathcal{T}$ w.r.t. $s_n^\bullet$ is* strongly correct *iff, 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) $s_i^\bullet$ is a term slice of s'_i, for all $i = 0, \ldots, n$.

Note that Definition 4.6 establishes that, for *every* concretization s'_0 of $s_0^\bullet$, which is compatible with the Boolean condition $B^\bullet$, there exists a concrete trace $\mathcal{T}'$ in which the rules involved in the sliced steps of $\mathcal{T}^\bullet$ can be applied again, at the same positions. Besides, each $s_i^\bullet$ is an appropriate term slice of s'_i, $i = 0, \ldots, n$.

Example 4.9

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)` $\overset{\texttt{debit}}{\rightarrow}$ `ac(A,25);credit(B,3)` $\overset{\texttt{credit}}{\rightarrow}$ `ac(A,28)`.

Let `ac(A,`$\bullet_1$`)` be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet$ be as follows:

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

Then, the trace slice $[\mathcal{T}^\bullet, \texttt{true}]$ is only weakly correct w.r.t. $\texttt{ac(A,}\bullet_1\texttt{)}$, whereas the trace slice $[\mathcal{T}^\bullet,\ \bullet_8 \geq \bullet_9]$ is strongly correct. Indeed, $\mathcal{T}^\bullet$ needs to be coupled with the compatibility condition $\bullet_8 \geq \bullet_9$ to ensure applicability of the `debit` rule. In other words, an instance $s^\bullet\sigma$ of $\texttt{ac(A,}\bullet_8\texttt{);debit(A,}\bullet_9\texttt{)}$ can be rewritten by the `debit` rule only if $\bullet_8\sigma \geq \bullet_9\sigma$.

Informally, given a slicing criterion $s_n^\bullet$ 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 of s_{i-1}. For each such rewrite step, the conditions of the applied rule are recursively processed in order to ascertain the meaningful information of s_{i-1} from s_i, 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 $(\mathit{Conf},\ \bullet\!\!\rightarrow)$ [20] 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 4.10 *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 $S^\bullet$, that records the computed term slice $s_i^\bullet$ of s_i*
- *a Boolean condition $B^\bullet$.*

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

Definition 4.11 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional 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. Given a set Conf of configurations, 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 $\mathit{slice\text{-}step}(V \rightarrow W,\ W^\bullet,\ B_W^\bullet)$ 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 $\langle s_0 \rightarrow^* s_n, s_n^\bullet,\ \mathit{true}\rangle$ 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

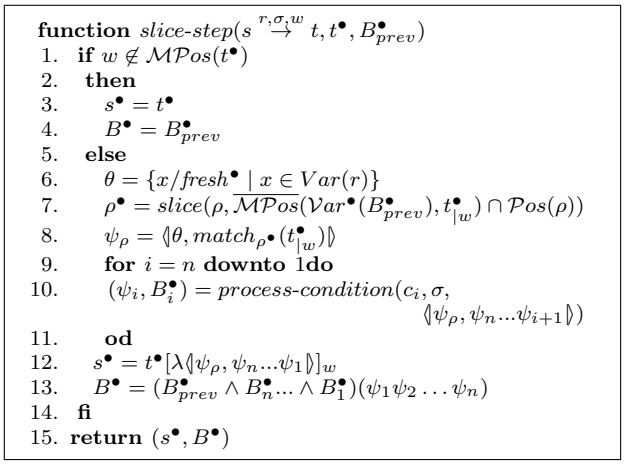

Figure 3: Backward step slicing function, with $r = (\lambda \to \rho \ \textit{if}\ C)$.

each term s_i by the corresponding term slice $s^\bullet_i$, $i = 0, \ldots, n$, in the original execution trace $s_0 \to^* s_n$. The algorithm additionally returns the accumulated compatibility condition $B^\bullet_0$ 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 4.12 (Backward trace slicing algorithm) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and let $\mathcal{T} : s_0 \to^* s_n$ be an execution trace in $\mathcal{R}$. Let $s^\bullet_n$ be a slicing criterion for $\mathcal{T}$. Then, the function backward-slicing is computed as follows:*

$$\textit{backward-slicing}(s_0 \to^* s_n, s^\bullet_n) = [s^\bullet_0 \to^* s^\bullet_n,\ B^\bullet_0]$$

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

$$\langle s_0 \to^* s_n,\ s^\bullet_n,\ true\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s^\bullet_{n-1},\ B^\bullet_{n-1}\rangle \bullet\!\!\to^* \langle s_0,\ s^\bullet_0,\ B^\bullet_0\rangle$$

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 $r = \lambda \to \rho \ \textit{if}\ C$[4], a term slice $t^\bullet$ of t, and a compatibility condition $B^\bullet_{prev}$. It delivers as outcome the term slice $s^\bullet$ and a new

[4]Since equations and axioms are both interpreted as rewrite rules in our formulation, notation $\lambda \to \rho \ \textit{if}\ C$ is often abused to denote rules as well as (oriented) equations and axioms.

compatibility condition $B^\bullet$. Within the algorithm *slice-step*, we use an auxiliary operator $\langle\!|\sigma_1, \sigma_2|\!\rangle$ that refines 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 (that are specified by $\bullet$-variables), and the bindings are incrementally refined as we (partially) solve the conditions of r.

Definition 4.13 (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:*

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

$$x\sigma = \begin{cases} x\sigma_2 & \text{if } x \in Dom(\sigma_1) \cap Dom(\sigma_2) \\ x\sigma_1\sigma_2 & \text{if } x \in Dom(\sigma_1) \setminus Dom(\sigma_2) \end{cases}$$

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 $\langle\!|\langle\!|\langle\!|\sigma_1, \sigma_2|\!\rangle, \ldots, \sigma_{n-1}|\!\rangle, \sigma_n|\!\rangle$. By abuse of notation, we also let $\langle\!|\sigma|\!\rangle$ denote the substitution σ.

Example 4.14

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 4.15

Consider the Maude specification of Example 2.1 and the following rewrite step μ: `ac(A,30);debit(A,5);credit(A,3)` $\overset{\texttt{debit}}{\rightarrow}$ `ac(A,25);credit(A,3)`. Let $\bullet_1$; `credit(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 $\bullet_1$; `credit(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 $\bullet_1$; `credit(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.

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

Figure 4: Condition processing function.

More specifically, we initially define the substitution $\theta = \{x/fresh^\bullet \mid x \in Var(r)\}$ 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 first compute the term slice $\rho^\bullet = slice(\rho, \overline{\mathcal{MP}os}(\mathcal{V}ar^\bullet(B^\bullet_{prev}), t^\bullet_{|w}) \cap \mathcal{P}os(\rho))$ that only records the meaningful symbols of the right-hand side ρ of the rule r w.r.t. the set of meaningful positions of the term slice $t^\bullet_{|w}$, including the positions of the $\bullet$-variables that appear in $B^\bullet_{prev}$ which are essential to correctly track back the compatibility condition computed so far. Then, by matching $\rho^\bullet$ with $t^\bullet_{|w}$, we generate a matcher $match_{\rho^\bullet}(t^\bullet_{|w})$ that extracts the meaningful information from $t^\bullet_{|w}$ that is used to refine θ. Formally, the computed refinement ψ_ρ of θ is given by $\psi_\rho = \langle\!|\theta, match_{\rho^\bullet}(t^\bullet_{|w})|\!\rangle$.

Example 4.16

Consider the rewrite theory in Example 2.1 together with the following rewrite step $\mu_{\texttt{debit}} : \texttt{ac(A,30);debit(A,5)} \overset{\texttt{debit}}{\rightarrow} \texttt{ac(A,25)}$ that involves the application (at position $w = \Lambda$) of the `debit` rule whose right-hand side is $\rho_{\texttt{debit}} = \texttt{ac(Id,nb)}$. Let $t^\bullet = \texttt{ac(A,}\bullet_1\texttt{)}$ be a term slice of `ac(A,25)` and $B^\bullet_{prev} = true$. 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 $\rho^\bullet_{\texttt{debit}} = slice(\texttt{ac(Id,nb)}, \overline{\mathcal{MP}os}(\emptyset, \texttt{ac(A,}\bullet_1\texttt{)}) \cap \mathcal{P}os(\texttt{ac(Id,nb)})) = \texttt{ac(Id,}\bullet_6\texttt{)}$. Finally, the refinement $\psi_{\rho_{\texttt{debit}}}$ of θ is given by

$$match_{\rho^\bullet_{\texttt{debit}}}(t^\bullet_{|w}) = match_{\texttt{ac(Id,}\bullet_6\texttt{)}}(\texttt{ac(A,}\bullet_1\texttt{)}) = \{\texttt{Id/A},\ \bullet_6/\bullet_1\}$$

and

$$\psi_{\rho_{\texttt{debit}}} = \langle\!|\theta, match_{\rho^\bullet_{\texttt{debit}}}(t^\bullet_{|w})|\!\rangle = \{\texttt{Id/A},\ \texttt{b}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{nb}/\bullet_5\}.$$

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

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 $(\psi_i,\ B_i^\bullet)$ 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 $B_i^\bullet$ 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 $B_i^\bullet$ delivered by *process-condition* and instantiating them with the composition of the computed refinements $\psi_1 \ldots \psi_n$. It is worth noting that *process-condition* handles rewrite expressions, equational conditions, and matching conditions differently. More specifically, the pair $(\psi_i,\ B_i)$ 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 $\mu : s \stackrel{r,\sigma,w}{\rightarrow} t$, 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 $\mathcal{T}_{int} = m\sigma \rightarrow^* p\sigma$, which is done by recursively invoking the function *backward-slicing* for execution trace slicing with respect to the slicing criterion given by slicing the term $p\sigma$ w.r.t. 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, $[(m\sigma)^\bullet \rightarrow^* (p\sigma)^\bullet,\ B^\bullet] = \mathit{backward\text{-}slicing}(m\sigma \rightarrow^* p\sigma,\ \mathit{slice}(p\sigma,\ \mathcal{MP}os(p\theta)))$. The result delivered by the function *backward-slicing* is a trace slice $(m\sigma)^\bullet \rightarrow^* (p\sigma)^\bullet$ with compatibility condition $B^\bullet$.

In order to deliver the final outcome for the matching condition $p := m$, we first compute the term slice $m^\bullet = \mathit{slice}(m, \overline{\mathcal{MP}os}(\mathcal{V}ar^\bullet(B^\bullet), (m\sigma)^\bullet) \cap \mathcal{P}os(m))$. Then, by matching $m^\bullet$ with $(m\sigma)^\bullet$, the algorithm yields a matcher ψ that extracts the meaningful information from $(m\sigma)^\bullet$ that will be used to refine the input substitution θ. Finally, the pair $(\psi,\ B^\bullet)$ is returned.

Example 4.17

Consider the rewrite step $\mu_{\texttt{debit}}$ of Example 4.16 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 $\mathcal{T}_{int} = \texttt{30} - \texttt{5} \rightarrow \texttt{25}$ [5]. By invoking the *backward-slicing* function with the slicing criterion given by $slice(25, \mathcal{MP}os(\bullet_1)) = \bullet_6$, the resulting trace slice is $[\bullet_6 \rightarrow \bullet_6, \ true]$. Then, the final outcome is given by computing $m^\bullet = slice(\texttt{30} - \texttt{5}, \overline{\mathcal{MP}os}(\emptyset, \bullet_6) \cap \mathcal{P}os(30-25)) = \bullet_7$ and $\psi = match_{\bullet_7}(\bullet_6)$, which is the substitution $\{\bullet_7/\bullet_6\}$.

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

– **Equational conditions.** During the execution of the rewrite step $\mu : s \overset{r,\sigma,w}{\rightarrow} t$, 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 substitution θ, in order to transfer only the meaningful information of $e\sigma$, computed so far, in e.

Example 4.18

Consider the refined substitution given in Example 4.17

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

together with the rewrite step $\mu_{\texttt{debit}}$ of Example 4.16 that involves the application of the `debit` rule. After processing the condition `b >= M` of `debit`, we deliver $B^\bullet = (\bullet_3 >= \bullet_4)$.

4.4. Correctness of Backward Trace Slicing

Given an execution trace $\mathcal{T} : s_1 \rightarrow \ldots s_n$ in a rewrite theory $\mathcal{R}$ and a slicing criterion $s_n^\bullet$, the call *backward-slicing*$(\mathcal{T}, s_n^\bullet)$ always produces a trace slice for $\mathcal{T}$ w.r.t. $s_n^\bullet$ that is weakly correct. This result is formalized in the following theorem which establishes the weak correctness of backward trace slicing.

[5] Note that the trace `30-5→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` → `c` as an ordinary rewrite step, we add the rule `a op b` ⇒ `c` to the considered rewrite theory.

Theorem 4.19 (weak correctness) *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} ... \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ be an execution trace in the rewrite theory $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \rightarrow ... \rightarrow s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing$(\mathcal{T}, s_n^\bullet)$ is a weakly correct trace slice of $\mathcal{T}$ w.r.t $s_n^\bullet$.*

Given an arbitrary rewrite theory, the outcome of *backward-slicing*$(\mathcal{T}, s_n^\bullet)$ might not be a strongly correct trace slice. For instance, when considering a rewrite rule that includes a matching condition $p := m$ or a rewrite expression $m \Rightarrow p$ such that the evaluation of m depends on the evaluation of some equational condition e, it could happen that e is not recorded in $B^\bullet$ because the evaluation of m is found to be irrelevant. In this scenario, we would lose the equational condition e that might be essential to enforce the strong correctness of the trace slice. Let us see an example.

Example 4.20

Consider the conditional rewrite theory $\mathcal{R}$ that contains the following rewrite rules:

$$r_1 : g(y) \rightarrow p(z) \text{ if } z := f(y)$$
$$r_2 : f(x) \rightarrow 5 \text{ if } x > 1$$

and the execution trace $\mathcal{T} : g(3) \overset{r_1}{\rightarrow} p(5)$. Note that the considered rewrite step implies the evaluation of the matching condition $z := f(y)$ that depends on rule r_2 which contains the equational condition $x > 1$.

Let $p(\bullet_1)$ be a slicing criterion for $\mathcal{T}$ which considers irrelevant the value computed by the matching condition $z := f(y)$. Then, *backward-slicing*$(\mathcal{T}, p(\bullet_1))$ yields the trace slice $[g(\bullet_2) \rightarrow p(\bullet_1), true]$, that is not strongly correct. Indeed, there exists the concretization $g(1)$ compatible with *true* that cannot be rewritten to $p(1)$ by using the rule r_1.

The next definition formalizes a class of rewrite theories for which strong correctness of backward trace slicing can be guaranteed, that is, the *backward-slicing* function only computes strongly correct trace slices. Roughly speaking, a rewrite theory satisfies the *separation property* if and only if it does not contain a mixture of equational conditions, rewrite expressions, and matching conditions. More formally,

Definition 4.21 *A conditional rewrite theory satisfies the separation property iff one of the two following clauses holds:*

- *all the conditional rewrite rules and equations of $\mathcal{R}$ only contain equational conditions;*
- *all the conditional rewrite rules and equations of $\mathcal{R}$ only contain matching conditions or rewrite expressions.*

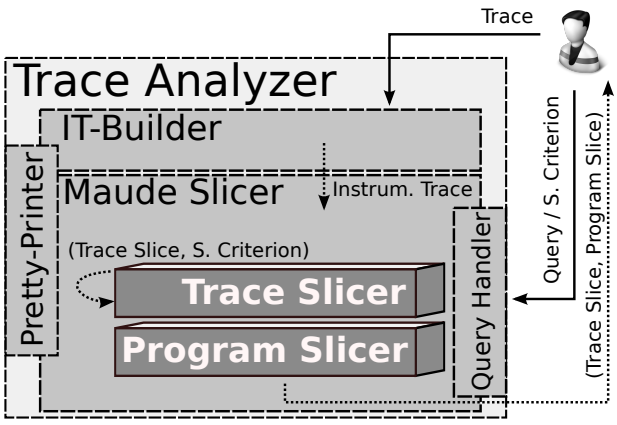

Figure 5: *i*JULIENNE architecture

Theorem 4.22 (strong correctness) *Let $\mathcal{R}$ be a conditional rewrite theory that satisfies the separation property. 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 $s_n^\bullet$ 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$(\mathcal{T}, s_n^\bullet)$ is a strongly correct trace slice of $\mathcal{T}$ w.r.t $s_n^\bullet$.*

The proofs of Theorem 4.19 and Theorem 4.22 can be found in Appendix A.

5. The *i*JULIENNE system

Our trace slicing methodology for conditional RWL computations has been implemented in the slicing-based trace analysis tool *i*JULIENNE which is available at [21]. The engine of *i*JULIENNE is written in Maude and consists of about 250 Maude function definitions (approximately 1.5K lines of source code). *i*JULIENNE is a stand-alone application (which can be invoked as a Maude command or used online through a Java web service) that allows the analysis of general rewrite theories that may contain (conditional) rules and equations, built-in operators, and algebraic axioms. The implementation uses meta-level capabilities and relies on the very efficient Maude system [16]. Actually, the current version of the Maude interpreter can do more than 3 million rewrites per second on state-of-the-art processors, and the Maude compiler can reach up to 15 million rewrites per second. Its user interface is based on the AJAX technology, which allows the Maude engine to be used through the WWW.

*i*JULIENNE generalizes and supersedes previous trace slicing tools such as the trace slicer for unconditional RWL theories described in [9], and the conditional slicer JULIENNE presented in [11].

The architecture of *i*JULIENNE, which is depicted in Figure 5, consists of four main modules named **IT-Builder**, **Maude Slicer**, **Query Handler**, and **Pretty-Printer**.

The **IT-Builder** is a pre-processor that obtains a suitable, instrumented trace meta-representation where all internal algebraic axiom applications are made explicit.

The **Maude Slicer** module provides incremental trace slicing and dynamic program slicing facilities. Both of these techniques are developed by using Maude reflection and meta-level functionality. On one hand, the *Trace Slicer* implements a greatly enhanced, incremental extension of the conditional backward trace slicing algorithm of [9, 10] in which the slicing criteria can be repeatedly refined and the corresponding trace slices are automatically obtained by simply discarding the pieces of information affected by the updates. Thanks to incrementality, trace slicing, analysis and debugging times are significantly reduced. On the other hand, the companion *Program Slicer* can be used to discard the program equations and rules that are not responsible for producing the set of target symbols in the observed trace state. Rather than simply glueing together the program equations and rules that are used in the simplified trace, it just delivers a program fragment that is proved to influence the observed result. In other words, not only are the unused program data and rules removed but the data and rules that are used in subcomputations that are irrelevant to the criterion of interest are also removed.

The way in which the slicing criteria are defined has been greatly improved in *i*Julienne. Besides supporting mouse click events that can select any information piece in the state, a **Query handling** facility is included that allows huge execution traces to be queried by simply providing a filtering pattern (the query) that specifies a set of symbols to monitor and also selects those states that match the pattern. A pattern language with wild cards ? and _ is used to identify (resp. discard) the relevant (resp. irrelevant) data inside the states.

Finally, the **Pretty-Printer** delivers a more readable representation of the trace (transformed back to sort String) that aims to favor better inspection and debugging within the Maude formal environment. Moreover, it provides the user with an advanced view where the irrelevant information can be displayed or hidden, depending on the interest of the user. This can also be done by automatically downgrading the color of those parts of the trace that contain subterms that are rooted by relevant symbols but that only have irrelevant children.

6. *i*Julienne at work

In general, conventional debugging is an inefficient and time-consuming approach for understanding 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 [22].

```
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.

*6.1. Debugging Maude programs with i*JULIENNE

In debugging, one is often interested in analyzing a particular 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 impact on an error or anomaly in the trace.

Basically, the idea is to feed *i*JULIENNE 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 and repeatedly refine the slicing criteria for program bugs to hunt, as shown in the following examples.

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.

Consider an execution trace $\mathcal{T}_{bank}$ in program `BANK_ERR` that starts in the initial state

Step	RuleName	Execution trace	Sliced trace
1	'Start	ac(A,50) ; ac(B,20) ; ac(C,20) ; ac(D,20) ; credit(A,10) ; credit(D,40) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4)	ac(*,*) ; ac(*,*) ; ac(*,20) ; ac(*,*) ; credit(*,*) ; credit(*,*) ; debit(*,50) ; debit(*,*) ; transfer(*,*,15) ; transfer(*,*,*) ; transfer(*,*,4)
2	unflattening	ac(B,20) ; ac(C,20) ; ac(D,20) ; credit(D,40) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,50) ; credit(A,10)	ac(*,*) **; ac(*,20) ;** ac(*,*) **;** credit(*,*) **; debit(*,50) ;** debit(*,*) **; transfer(*,*,15) ;** transfer(*,*,*) **; transfer(*,*,4) ;** ac(*,*) **;** credit(*,*)
3	credit	ac(B,20) ; ac(C,20) ; ac(D,20) ; credit(D,40) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,60)	ac(*,*) ; ac(*,20) ; ac(*,*) ; credit(*,*) ; debit(*,50) ; debit(*,*) ; transfer(*,*,15) ; transfer(*,*,*) ; transfer(*,*,4) ; ac(*,*)
4-5	flat-unflat	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,20) ; credit(D,40)	ac(*,*) **;** ac(*,*) **; ac(*,20) ; debit(*,50) ;** debit(*,*) **; transfer(*,*,15) ;** transfer(*,*,*) **; transfer(*,*,4) ;** ac(*,*) **;** credit(*,*)
6	credit	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,60)	ac(*,*) ; ac(*,*) ; ac(*,20) ; debit(*,50) ; debit(*,*) ; transfer(*,*,15) ; transfer(*,*,*) ; transfer(*,*,4) ; ac(*,*)
7-8	flat-unflat	ac(A,60) ; ac(B,20) ; ac(D,60) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(C,20) ; debit(C,50)	ac(*,*) **;** ac(*,*) **;** ac(*,*) **;** debit(*,*) **; transfer(*,*,15) ;** transfer(*,*,*) **; transfer(*,*,4) ; ac(*,20) ; debit(*,50)**
9	debitERR	ac(A,60) ; ac(B,20) ; ac(D,60) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(C,- 30)	ac(*,*) ; ac(*,*) ; ac(*,*) ; debit(*,*) ; transfer(*,*,15) ; transfer(*,*,*) ; transfer(*,*,4) ; ac(*,- 30)
10-11	flat-unflat	ac(A,60) ; ac(B,20) ; ac(C,- 30) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,60) ; debit(D,5)	ac(*,*) **;** ac(*,*) **; ac(*,- 30) ; transfer(*,*,15) ;** transfer(*,*,*) **; transfer(*,*,4) ;** ac(*,*) **;** debit(*,*)
12	debitERR	ac(A,60) ; ac(B,20) ; ac(C,- 30) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,55)	ac(*,*) ; ac(*,*) ; ac(*,- 30) ; transfer(*,*,15) ; transfer(*,*,*) ; transfer(*,*,4) ; ac(*,*)
13-14	flat-unflat	ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,60) ; ac(C,- 30) ; transfer(A,C,15)	ac(*,*) **;** ac(*,*) **;** transfer(*,*,*) **; transfer(*,*,4) ;** ac(*,*) **; ac(*,- 30) ; transfer(*,*,15)**
15	transfer	ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,45) ; ac(C,- 15)	ac(*,*) ; ac(*,*) ; transfer(*,*,*) ; transfer(*,*,4) ; ac(*,*) ; ac(*,- 15)
16-17	flat-unflat	ac(B,20) ; ac(C,- 15) ; transfer(B,C,4) ; ac(A,45) ; ac(D,55) ; transfer(A,D,20)	ac(*,*) **; ac(*,- 15) ; transfer(*,*,4) ;** ac(*,*) **;** ac(*,*) **;** transfer(*,*,*)
18	transfer	ac(B,20) ; ac(C,- 15) ; transfer(B,C,4) ; ac(A,25) ; ac(D,75)	ac(*,*) ; ac(*,- 15) ; transfer(*,*,4) ; * ; *
19-20	flat-unflat	ac(A,25) ; ac(D,75) ; ac(B,20) ; ac(C,- 15) ; transfer(B,C,4)	* **;** * **;** ac(*,*) **; ac(*,- 15) ; transfer(*,*,4)**
21	transfer	ac(A,25) ; ac(D,75) ; ac(B,16) ; ac(C,- 11)	* ; * ; * ; ac(*,- 11)
22	flattening	ac(A,25) ; ac(B,16) ; ac(C,- 11) ; ac(D,75)	* **;** * **; ac(*,- 11) ;** *
Total size:		1689	295
Reduction: 83%			

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

```
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 client `C`, which is a clear symptom of malfunction of the `BANK_ERR` specification, if we assume that balances must be non-negative numbers according to the semantics intended by the programmer. Therefore, we execute *i*JULIENNE on the trace $\mathcal{T}_{bank}$ w.r.t. the slicing criterion that observes the term $\texttt{ac}(\texttt{C}, -11)$ in order to determine the cause of such a negative balance. The output delivered by *i*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 83%, 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 `9` in the trace slice $\mathcal{T}^{\bullet}_{bank}$ has been obtained by reducing the term `debit(C, 50)` by means of the rule `debitERR` even though the current 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.

The following example illustrates how debugging activities can be greatly improved by using the incremental trace slicing facility provided by *i*JULIENNE.

Example 6.2

Consider the Maude program `BLOCKS-WORLD` of Figure 7, which is a faulty mutation of one of the most popular planning problems in artificial intelligence.We assume that there are some blocks, placed on a table, that can be moved by means of a robot arm; the goal of the robot arm is to produce one or more vertical stacks of blocks. In our specification, which is shown in the Maude module `BLOCKS-WORLD` of Figure 7, we define a Blocks World system with three different kinds of blocks that are defined by means of the operators `a, b`, and `c` of sort `Block`. Different blocks have different sizes that are described by using the unary operator `size`. We also consider some operators that formalize block and robot arm properties whose intuitive meanings are given in the accompanying program comments.

The states of the system are modeled by means of associative and commutative lists of properties of the form $\texttt{prop}_1\&\texttt{prop}_2\&\ldots\&\texttt{prop}_n$, which describe any possible configuration of the blocks on the table as well as the status of the robot arm.

The system behavior is formalized by four, simple rewrite rules that control the robot arm. Specifically, the `pickup` rule describes how the robot arm grabs a block from the table, while `putdown` rule corresponds to the inverse move. The `stack` and `unstack` rules respectively allow the robot arm to drop one block on top of another block and to remove a block from the top of a stack. Note that the conditional `stack` rule forbids a given block $\texttt{B}_1$ from being piled on a block $\texttt{B}_2$ if the size of $\texttt{B}_1$ is greater than the size of $\texttt{B}_2$.

Barely perceptible, the Maude specification of Figure 7 fails to provide a correct Blocks World implementation. By using the `BLOCKS-WORLD` module, it is indeed possible

```
mod BLOCKS-WORLD is inc INT .
  sorts Block Prop State .
  subsort Prop < State .
  ops a b c : -> Block .
  op table : Block -> Prop .      *** block is on the table
  op on : Block Block -> Prop .   *** first block  is on the second block
  op clear : Block -> Prop .      *** block is clear
  op hold : Block -> Prop .       *** robot arm holds the block
  op empty : -> Prop .            *** robot arm is empty
  op _&_ : State State -> State [assoc comm] .
  op size : Block -> Nat .
  vars X Y : Block .

  eq [sizeA] : size(a) = 1 .
  eq [sizeB] : size(b) = 2 .
  eq [sizeC] : size(c) = 3 .

  rl [pickup] :  clear(X) & table(X) => hold(X) .
  rl [putdown] : hold(X) => empty & clear(X) & table(X) .
  rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
  crl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) if size(X) < size(Y) .
endm
```

Figure 7: BLOCKS-WORLD faulty Maude specification.

to derive system states that represent erroneous configurations. For instance, the initial state

$$\mathtt{s_i} = \mathtt{empty}\ \&\ \mathtt{clear(a)}\ \&\ \mathtt{table(a)}\ \&\ \mathtt{clear(b)}\ \&\ \mathtt{table(b)}\ \&\ \mathtt{clear(c)}\ \&\ \mathtt{table(c)}$$

describes a simple configuration where the robot arm is empty and there are three blocks a, b, and c on the table. It can be rewritten in 7 steps to the state

$$\mathtt{s_f} = \boxed{\mathtt{empty}}\ \&\ \boxed{\mathtt{empty}}\ \&\ \mathtt{table(b)}\ \&\ \mathtt{table(c)}\ \&\ \mathtt{clear(a)}\ \&\ \mathtt{clear(c)}\ \&\ \boxed{\mathtt{on(a,b)}}$$

which clearly indicates a system anomaly since it shows the existence of two empty robot arms!

To find the cause of this wrong behavior, we feed *i*Julienne with the faulty rewrite sequence $\mathcal{T} = \mathtt{s_i} \rightarrow^* \mathtt{s_f}$, and we initially slice $\mathcal{T}$ w.r.t. the slicing criterion that observes the two anomalous occurrences of the empty property and the stack $\mathtt{on(a,b)}$ in State $\mathtt{s_f}$. This task can be easily performed in *i*Julienne by first highlighting the terms that we want to observe in State $\mathtt{s_f}$ with the mouse pointer and then starting the slicing process. *i*Julienne yields a trace slice that simplifies the original trace by recording only those data that are strictly needed to produce the considered slicing criterion. Also, it automatically computes the corresponding program slice, which consists of the equations defining the size operator together with the pickup and stack rules (see Figure 8). This allows us to deduce that the malfunction is located in one or more rules and equations that are included in the program slice.

The generated trace slice is then browsed backwards using the *i*Julienne's navigation facility in search of a possible explanation for the wrong behavior. During this

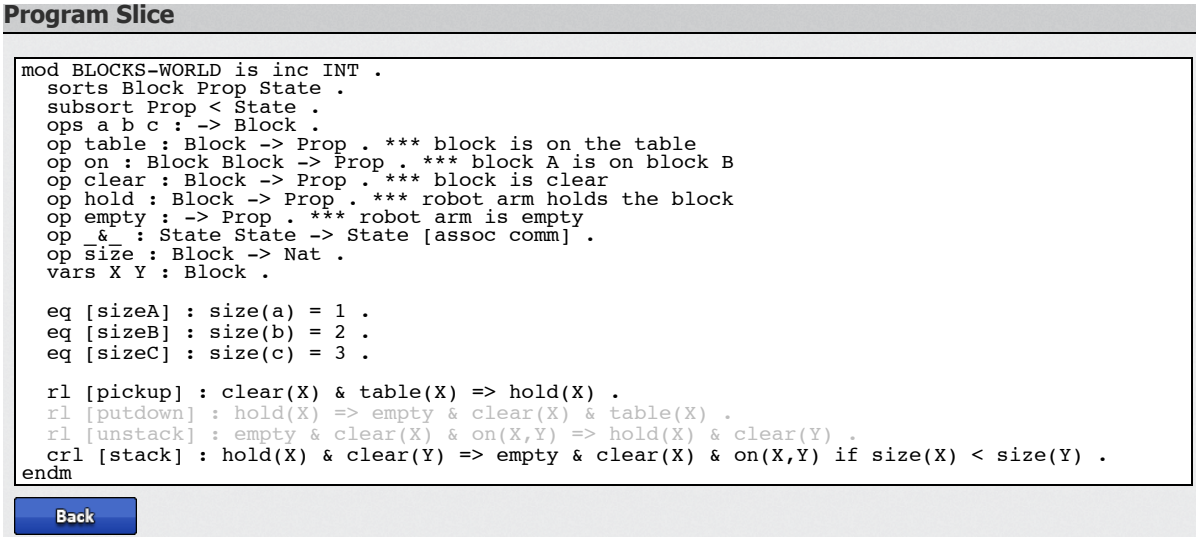

Figure 8: Program slice computed w.r.t. the slicing criterion `empty`, `empty`, `on(a,b)`.

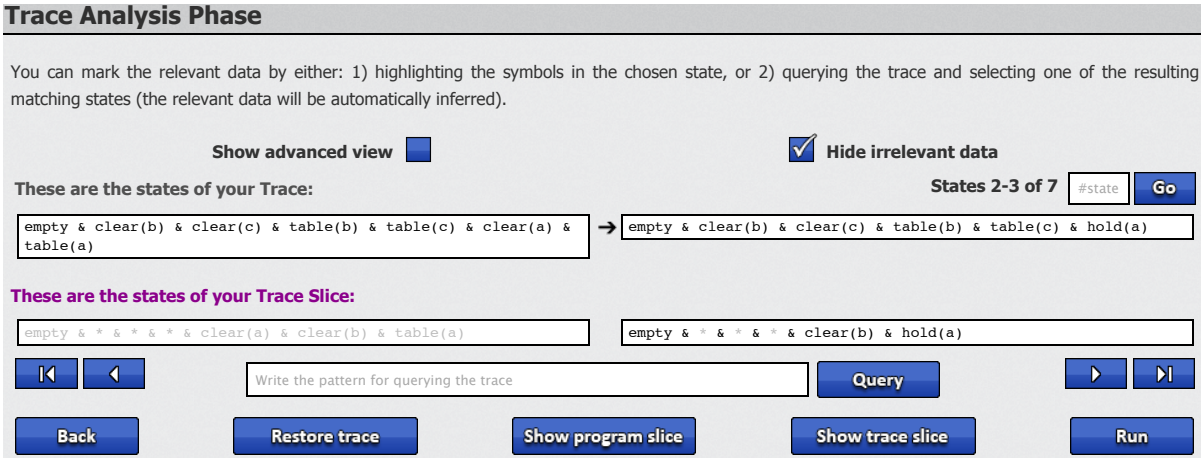

Figure 9: Navigation through the trace slice of the *Blocks World* example.

phase, we focus our attention on State 3 (Figure 9), which is inconsistent since it models a robot arm that is holding block `a` and is empty at the same time. Therefore, we decide to further refine the trace slice by incrementally applying backward trace slicing to State 3 w.r.t. the slicing criterion `hold(a)`. This way we achieve a supplementary reduction of the previous trace slice in which we can easily observe that `hold(a)` only depends on the `clear(a)` and `table(a)` properties (see Figure 10). Furthermore, the computed program slice includes the single `pickup` rule (see Figure 11). Thus, we can conclude that:

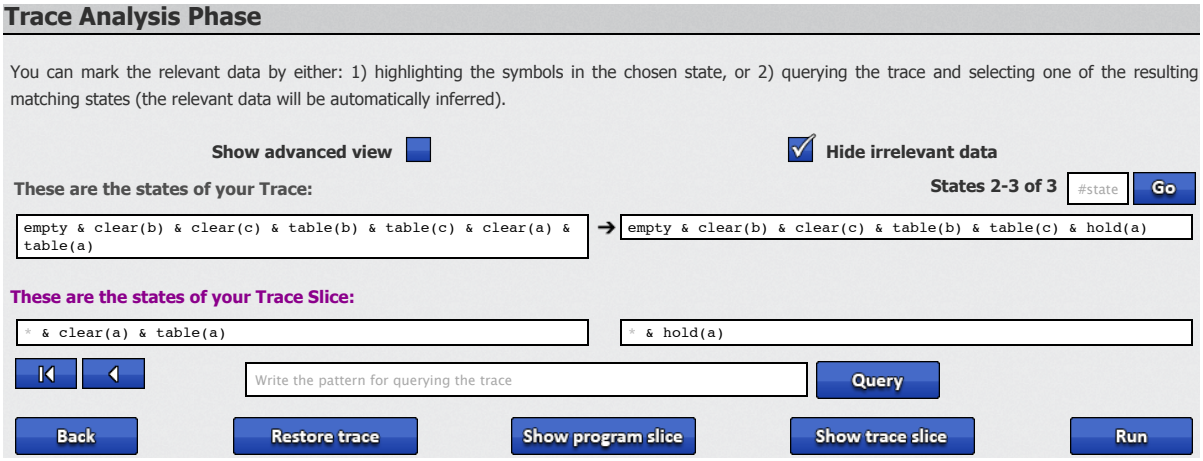

Figure 10: Navigation though the refined trace slice of the *Blocks World* example.

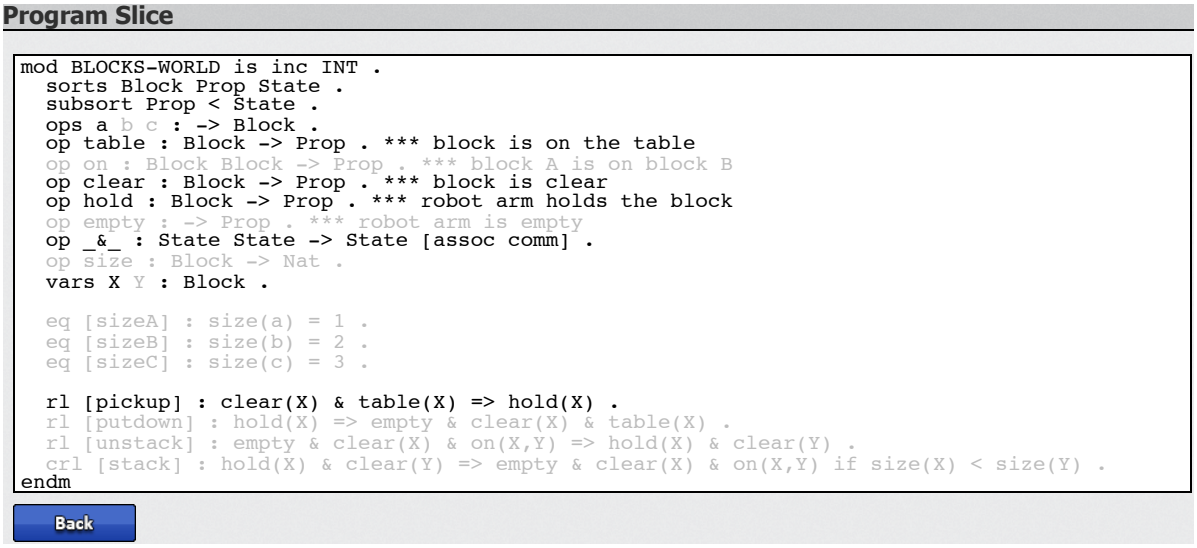

Figure 11: Program slice computed w.r.t. the slicing criterion `hold(a)`.

1. the malfunction is certainly located in the `pickup` rule (since the computed program slice only contains that rule);
2. the `pickup` rule does not depend on the status of the robot arm (this is witnessed by the fact that `hold(a)` only relies on the `clear(a)` and `table(a)` properties);
3. by 1 and 2, we can deduce that the `pickup` rule is incorrect, as it never checks the emptiness of the robot arm before grasping a block.

A possible fix of the detected error consists in including the `empty` property in the

left-hand side of the `pickup` rule, which enforces the robot arm to always be idle before picking up a block. The corrected version of the rule is hence as follows:

```
rl [pickup] :  empty & clear(X) & table(X) => hold(X) .
```

---

*6.2. Trace querying with i*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 [23] 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 *i*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 (strong) 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 6.3

The Maude specification of Figure 12, inspired by a similar one in [24], 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`.

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. *i*JULIENNE generates the trace slice $\mathcal{T}^{\bullet}_{minmax}$, given in Table 2, whose compatibility condition is $\mathtt{C}^{\bullet} = \bullet_1 > 0 \wedge \bullet_3 > 0$. 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

```
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 .
   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) .
   rl [minmax2] : minmax(N ; L) => PAIR(Min(N,1st(minmax(L))) , Max(N,2nd(minmax(L)))) .
endm
```

Figure 12: Maude specification of the `minmax` function

Step	RuleName	Execution trace	Sliced trace
1	'Start	minmax(4 ; 7 ; 0)	minmax(*1 ; *3 ; 0)
2	unflattening	minmax(4 ; 7 ; 0)	**minmax(***1* **; ***3* **; 0)**
3	minmax2	PAIR(Min(4,1st(minmax(7 ; 0))),Max(4,2nd(minmax(7 ; 0))))	**PAIR(Min(***1***,1st(minmax(***3* **; 0))),***0***)**
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(***1***,1st(PAIR(Min(***3***,1st(minmax(0))),***10***))),***0***)**
5	1st	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(*1,Min(*3,1st(minmax(0)))),*0)
6	minmax2	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))))	PAIR(Min(*1,Min(*3,1st(minmax(0)))),*0)
7	2nd	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(***1***,Min(***3***,1st(minmax(0)))),***0***)**
8	minmax1	PAIR(Min(4,Min(7,1st(PAIR(0,0)))),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(***1***,Min(***3***,1st(PAIR(0,***6***))),***0***)**
9	1st	PAIR(Min(4,Min(7,0)),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(***1***,Min(***3***,0)),***0***)**
10	Min1	PAIR(Min(4,0),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(***1***,0),***0***)**
11	Min1	PAIR(0,Max(4,Max(7,2nd(minmax(0)))))	PAIR(0,*0)
12	minmax1	PAIR(0,Max(4,Max(7,2nd(PAIR(0,0)))))	PAIR(0,*0)
13	2nd	PAIR(0,Max(4,Max(7,0)))	PAIR(0,*0)
14	Max1	PAIR(0,Max(4,7))	PAIR(0,*0)
15	Max2	PAIR(0,7)	**PAIR(0,***0***)**
Compatibility condition:		*1 > 0 and *3 > 0	
Total size:		679	219
Reduction: 67%			

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

slicing technique, we can state that for every concrete instance $\mathtt{L_c}$ of the partially-defined input $\bullet_1;\bullet_3;\mathtt{0}$ that meets the compatibility condition $\mathtt{C}^\bullet$, the minimum computed by the call $\mathtt{minmax}(\mathtt{L_c})$ will be $\mathtt{0}$.

The `MINMAX` program is paradigmatic of a strategy known as introduction of mutual recursion, which separates a single function yielding pairs into the pair of its component functions (slices) [25]. Thus in a sense, the same effect (trace slicing) could be obtained for this example by slicing the source code in Maude before trace inspection.

In the following example, the trace querying capabilities of *i*Julienne are used to simplify the counter-examples traces provided by Web-TLR [5], a RWL-based tool that allows real-size web applications to be formally specified and verified by using the built-in Maude model-checker, thus facilitating their analysis. 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 [26]), which is a temporal logic specifically designed to model-check rewrite theories. A property is refuted by the LTLR model-checker by issuing a counter-example in the form of a rewrite sequence that reveals some undesired, erroneous behavior. Our example reproduces a typical trace analysis session that operates on the Maude specification of a complex webmail application checked with Web-TLR.

Example 6.4

Consider the webmail application written in Maude given in [5] that models both server-side aspects (e.g., web script evaluations, database interactions) and browser-side features (e.g., forward/backward navigation, web page refreshing, window/tab openings). The web application behavior is formalized by using rewrite rules of the form $[\mathsf{label}] : \mathsf{webState} \Rightarrow \mathsf{webState}$, where webState is a triple that we represent with the following operator $\_\|\_\|\_ : (\mathsf{Browsers} \times \mathsf{Message} \times \mathsf{Server}) \rightarrow \mathsf{webState}$ that can be interpreted as a system shot that captures the current configuration of the active browsers (i.e., the browsers currently using the webmail application) together with the channel through which the browsers and the server communicate via message-passing. An execution trace specifies a sequence of webState transitions that represents a possible execution of the webmail application.

The trace slicing session starts by loading the Maude webmail specification, together with the execution trace to be analyzed, into the *i*Julienne trace analyzer. The trace can be directly pasted in the input form or uploaded from a trace file that was written off-line. It can also be dynamically computed by the system (using Maude meta-search capabilities) by introducing the initial and final states of the trace. In Figure 13, we directly fed *i*Julienne with an execution trace that represents a counter-example that was automatically generated by the Maude LTLR model-checker. The considered trace consists of 97 states, each of which has about 5.000 characters.

The aim of our analysis is to extract the navigation path of a possible malicious user `idA` within the web application from the execution trace. This is particularly hard to

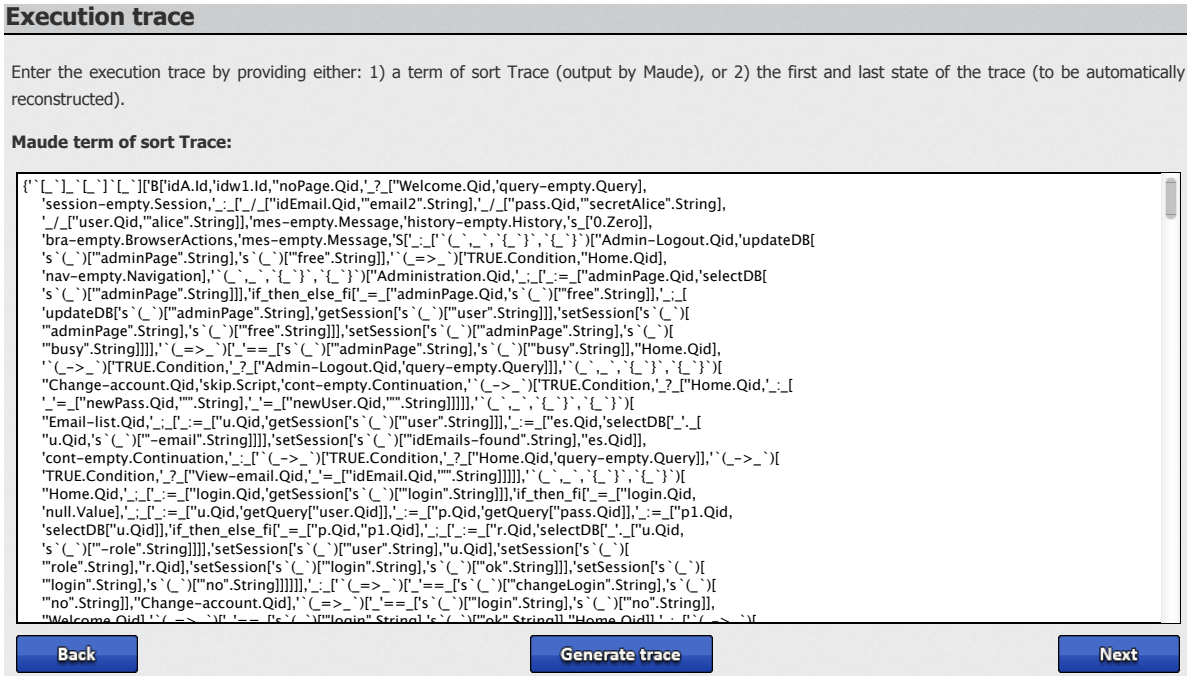

Figure 13: Loading the webmail execution trace.

perform by hand since the trace is extremely large and system states contain a huge amount of data.

Therefore, we decide to slice the trace with *i*Julienne in order to isolate only the web interactions related to user `idA`, getting rid of all the remaining unrelated information. To this end, we define the query `B`(`idA`, _, ?, _, _, _, _, _, _), which allows us to select all states in the trace that contain a browser data structure that is associated with user `idA`. Note that only the third argument of the browser data structure, which corresponds to the web page displayed on the browser, is declared relevant in the query (i.e., it is marked by the card ?). Indeed, we are interested in tracking only the web pages visited by the user `idA`. The remaining information (such as script evaluation, and web interactions with other users) is not pinpointed and, hence, will be systematically removed by the slicing tool which facilitates comprehension. By running the query, *i*Julienne computes the outcome shown in Figure 14, which delivers the states that satisfy the query. More concretely, the first 20 states of the trace do match the query, while the remaining 77 states do not include any web interaction with the user `idA` and thus do not need to be inspected.

Now, since we are interested in observing the whole navigation history of user `idA`, we select the last state in the trace that matches the query, namely State 20, and we apply backward trace slicing on that state in order to generate a simplified view of the first 20 states of the trace. Note that the slicing criterion is automatically computed by

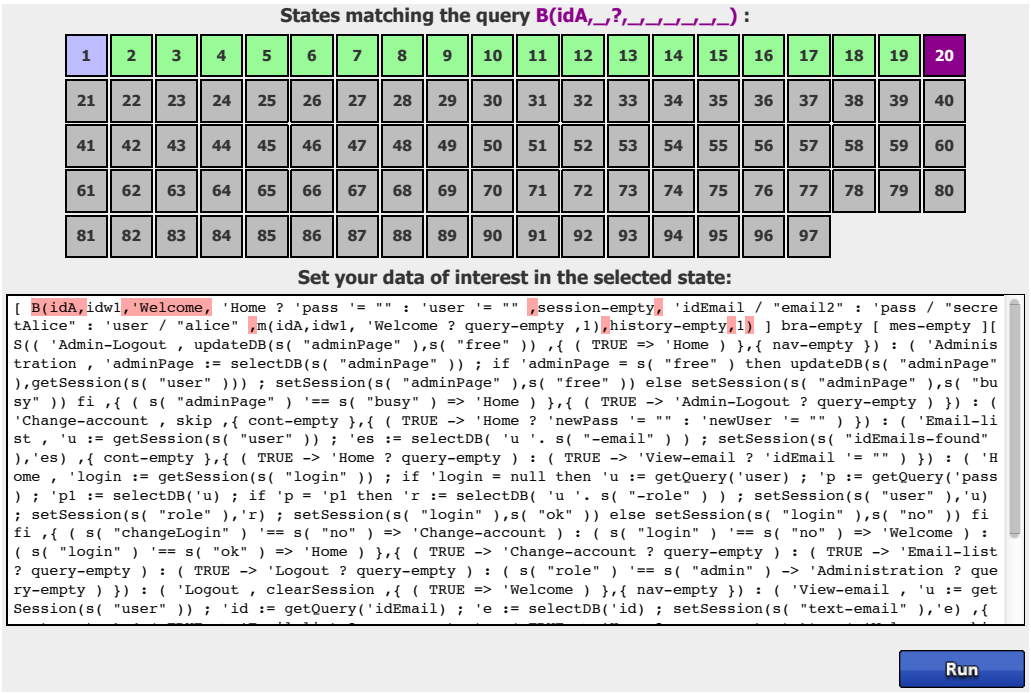

Figure 14: Querying the webmail trace w.r.t. B(idA, _, ?, _, _, _, _, _, _)

*i*JULIENNE by extracting the data matching the query from State 20 (specifically, the target symbols B, idA, and the current browser web page Welcome).

After pressing the *Run* button, we get a browsable trace slice (see Figure 15) where each state of the trace slice is purged of the irrelevant data w.r.t. the slicing criterion, and all the rewrite steps that do not affect the observed data are marked as irrelevant and are simply removed from the slice, which further reduces its size. The reduction rate achieved w.r.t. the original trace reaches an impressive 91%; Specially, 88 of the 97 states were found to be redundant with regard to the selected slicing criterion. This makes the trace slice easy to analyze by hand. Actually, by navigating through the trace slice, it can be immediately observed that the malicious user idA visits the Login page, succeeds to log onto the webmail system and enters the webmail Welcome page.

6.3. Dynamic program slicing

As we illustrated in Figures 8 and 11, for the BLOCKS-WORLD example, by running *i*JULIENNE one is able to obtain not only a more compact and focused trace that corresponds to the execution of the program, but also to uncover statement dependencies that connect computationally related parts of the program. Hence a single walk over such dependencies is sufficient to implement a form of dynamic program slicing.

Program slicing [22] 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

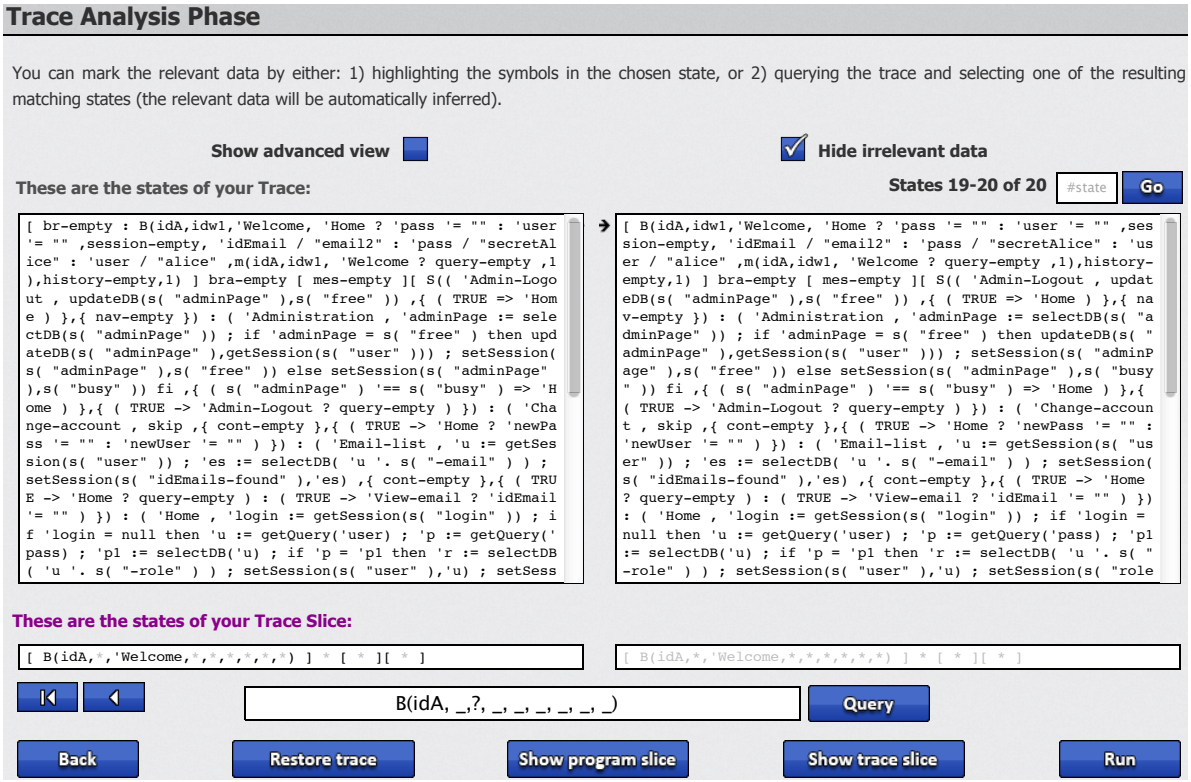

Figure 15: webmail trace slice after querying the trace

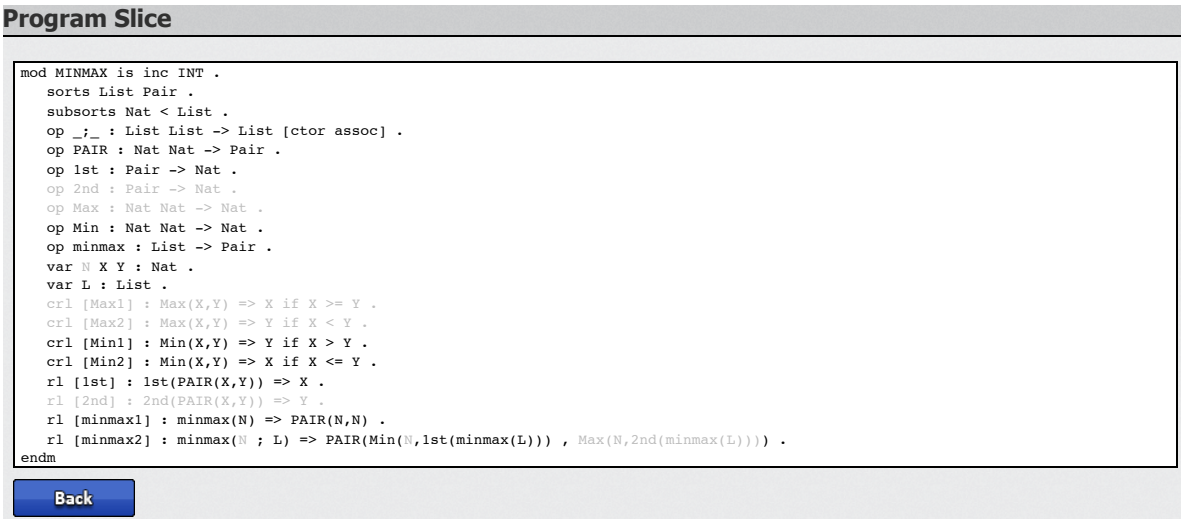

Figure 16: `MINMAX` program slice computed w.r.t. the query `PAIR`(?, _)

of a subset of the statements of the original program, sometimes with the additional constraint that a slice must be a syntactically valid, executable program. Relevant applications of slicing include software maintenance, optimization, program analysis, and information flow control. An important distinction holds 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 particular execution of the program (i.e., a given execution trace) [14], hence it only reflects the actual dependencies 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 path. 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 fragments in a given trace slice.

Example 6.5

Consider the trace slice $\mathcal{T}^{\bullet}_{minmax}$ of Example 6.3 for the execution trace

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

w.r.t. the query $\texttt{PAIR}(?,\_)$. Since operations `2nd` and `Max` are never used in $\mathcal{T}^{\bullet}_{minmax}$, *i*JULIENNE generates a dynamic program slice of the `MINMAX` Maude module by hiding

- the unneeded operator and variable declarations in the program signature;
- the rule definitions of the functions `2nd` and `Max`;
- the function calls to `2nd` and `Max` in the right-hand side of the `minmax2` rule.

Figure 16 shows the resulting dynamic program slice.

6.4. Experimental evaluation

*i*JULIENNE is the first slicing-based, incremental trace analysis tool for Rewriting Logic that greatly reduces the size of the computation traces and can make their analysis feasible even for complex, real-size applications. *i*JULIENNE conveys an incremental slicing approach where the user can refine the slicing criteria and then the extra redundancies (i.e., the difference of the slices) are automatically done away with. It is important to note that our trace analyzer does not remove any information that is relevant, independently of the skills of the user. We have experimentally evaluated our tool in several case studies that are available at the *i*JULIENNE web site [21] 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 *i*JULIENNE on several execution traces of increasing complexity. More precisely, we have considered

Example trace	Original trace size	Slicing criterion	Sliced trace size	% reduction	% reduction by changing criterion
FTCP.$\mathcal{T}_1$	2054	FTCP.$\mathcal{T}_1$.O_1	294	85.69%	97.89%
		FTCP.$\mathcal{T}_1$.O_2	316	84.62%	97.30%
FTCP.$\mathcal{T}_2$	1286	FTCP.$\mathcal{T}_2$.O_1	135	89.40%	98.11%
		FTCP.$\mathcal{T}_2$.O_2	97	92.46%	99.01%
Maude-NPA.$\mathcal{T}_1$	21265	Maude-NPA.$\mathcal{T}_1$.O_1	2249	89.42%	98.12%
		Maude-NPA.$\mathcal{T}_1$.O_2	2261	89.36%	98.03%
Maude-NPA.$\mathcal{T}_2$	34681	Maude-NPA.$\mathcal{T}_2$.O_1	3015	91.30%	99.08%
		Maude-NPA.$\mathcal{T}_2$.O_2	3192	90.79%	98.84%
web-TLR.$\mathcal{T}_1$	38983	web-TLR.$\mathcal{T}_1$.O_1	2045	94.75%	99.28%
		web-TLR.$\mathcal{T}_1$.O_2	2778	92.87%	99.14%
web-TLR.$\mathcal{T}_2$	69491	web-TLR.$\mathcal{T}_2$.O_1	8493	87.78%	97.99%
		web-TLR.$\mathcal{T}_2$.O_2	5034	92.76%	99.05%
SmallKb.$\mathcal{T}_1$	4149	SmallKb.$\mathcal{T}_1$.O_1	459	88.94%	98.08%
		SmallKb.$\mathcal{T}_1$.O_2	389	90.62%	98.26%
SmallKb.$\mathcal{T}_2$	5718	SmallKb.$\mathcal{T}_2$.O_1	419	92.67%	98.89%
		SmallKb.$\mathcal{T}_2$.O_2	618	89.19%	98.15%
		% reduction average		90.16%	98.45%

Table 3: Incremental slicing benchmarks.

- 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 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 [27], 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 [28]. 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.

- two counter-examples produced by model-checking a real-size webmail application specified in Web-TLR. The chosen slicing criteria allow 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).

- two execution traces generated by the Pathway Logic Maude implementation. Pathway Logic [29] is a RWL-based approach to modeling biological entities and

processes that formalizes the ordinary models that biologists commonly use to explain biological processes. Roughly speaking, Pathway logic models are Maude executable specifications whose execution traces provide a rewriting-based description of metabolic pathways. The traces that we consider in our experiments model responses to signal stimulation in epithelial-like cells. The chosen criteria allow one to detect cause and effect relations (e.g., the signal responsible for the production of an observed chemical) and select only those chemical reactions that are involved in actions of interest (e.g., protein complexing, phosphorylation).

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, not only do the trace slices that we obtained show impressive reduction rates (ranging from ∼85% to ∼95%), but we were also even able to strikingly improve these rates by an average of 8.5% (ranging from ∼97% to ∼99%) by using *incremental slicing*. In most cases, the delivered trace slices were cleaned enough to be easily analyzed, and we noted an increase in the effectiveness of the analysis processes. Other benchmark programs we have considered are available at the *i*JULIENNE web site.

With regard to the time required to perform the analyses, our implementation is rather 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, 13, 30, 31], and all of them apply to unconditional systems. The techniques in [9, 13, 30] rely on a labeling relation on symbols that allows data content to be traced back within the computation; this is achieved in [31] by formalizing a notion of dynamic dependence among symbols by means of contexts. In [13, 30], 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 dependency among the symbols involved in an execution step by using a procedure (based on [30]) that dynamically labels the calls (terms) involved in the steps. This paper describes 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 *i*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.

HaSlicer [32] is a prototype of a slicer for functional programs written in Haskell that is used for the identification of possible coherent components from (functional) monolithic legacy code. Both backward and forward dependency slicing are covered by HaSlicer, which is proposed as a support tool for the software architect to manually improve program understanding, and automatically discover software components. The latter is particularly useful as an architecture understanding technique in earlier phases of a re-engineering process.

Spyder [33] is a prototype debugger for C 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 and HaSlicer, our technique is based on *trace slicing* rather than *program slicing*, and needs much less storage to perform flow-back analysis, as it requires neither the construction of data and control dependency graphs nor the creation of an execution history.

The Haskell interactive debugger Hat [34] 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 [35] to help perform Java program analysis (e.g., dynamic *program slicing* [36]) 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, and is useful for execution trace analyses of sophisticated rewrite theories that may include conditional equations,

equational axioms, and rules. The technique has been implemented in the *i*JULIENNE system. *i*JULIENNE reveals that the technique does not come at the expense of performance with respect to existing Maude tools. This makes *i*JULIENNE attractive for Maude users, especially taking into account that program debugging and trace analysis in Maude is tedious with current state-of-the-art tools. 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 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 analyses, including the *incremental* algorithm that supports stepwise refinements of the trace slice.

The Maude system currently supports two different approaches for debugging Maude programs: the internal debugger described in [16] (Chap. 22) and the declarative debugger of [37]. The former 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 [38] 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) [37]. 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'?" As 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. In order to achieve this, we intend to develop a trace slicing 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 [39] which is concerned with monitoring and analysis of system executions. More specifically, we can consider a programming language $\mathcal{L}$ which is provided with 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 [40], and hence backward trace slicing can be used to analyze such semantics-driven executions w.r.t. a reference specification that monitors critical data. This way, runtime verification might be semantically grounded in our setting, while it is commonly offhacked in more traditional approaches by means of program instrumentation.

Acknowledgement

We would like to acknowledge the participation of Julia Sapiña in the implementation of *i*JULIENNE.

References

[1] F. Chen, G. Roşu, Parametric trace slicing and monitoring, in: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '09, Springer-Verlag, Berlin, Heidelberg, 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 Programming 81 (2012) 782–815.

[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 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 theories, in: Proc. of 18th Int'l Symposium on Formal Methods FM 2012, LNCS, Springer, 2012, pp. 28–32.

[12] M. Alpuente, D. Ballis, F. Frechina, J. Sapina, Slicing-based trace analysis of rewriting logic specifications with *i*JULIENNE, in: Proc. of 22nd European Symposium on Programming ESOP 2013, LNCS, Springer, 2013, pp. 121–124.

[13] Terese, Term Rewriting Systems, Cambridge University Press, Cambridge, UK, 2003.

[14] H. Agrawal, J. R. Horgan, Dynamic program slicing, in: PLDI, pp. 246–256.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60-61 (2004) 17–139.

[21] The *i*JULIENNE web site, 2013. Available at: `http://safe-tools.dsic.upv.es/iJulienne/`.

[22] M. Weiser, Program slicing, in: Proceedings of the 5th international conference on Software engineering, ICSE '81, IEEE Press, Piscataway, NJ, USA, 1981, pp. 439–449.

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

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

[25] G. Villavicencio, J. N. Oliveira, Reverse Program Calculation Supported by Code Slicing, in: WCRE, IEEE Computer Society, 2001, pp. 35–46.

[26] 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.

[27] 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.

[28] 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.

[29] C. L. Talcott, Pathway logic, in: SFM, LNCS, Springer, 2008, pp. 21–53.

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

[31] 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.

[32] N. F. Rodrigues, L. S. Barbosa, Component identification through program slicing, Electr. Notes Theor. Comput. Sci. 160 (2006) 291–304.

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

[34] 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.

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

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

[37] A. Riesco, A. Verdejo, N. Martí-Oliet, A complete declarative debugger for maude, in: Algebraic Methodology and Software Technology - 13th Int'l Conference, AMAST 2010, volume 6486 of *LNCS*, Springer, 2010, pp. 216–225.

[38] E. Y. Shapiro, Algorithmic Program Diagnosis, in: Conference Record of Ninth Annual ACM Symposium on Principles of Programming Languages, POPL'82, ACM Press, 1982, pp. 299–308.

[39] 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 *LNCS*, Springer, 2010.

[40] A. Farzan, F. Chen, J. Meseguer, G. Rosu, Formal analysis of Java programs in JavaFAN, in: Proc. of 16th International Conference on Computer Aided Verification (CAV 2004), volume 3114 of *LNCS*, Springer, 2004, pp. 501–505.

Appendix A. Proofs of Theorem 4.19 and Theorem 4.22

Throughout this section, we assume that all execution traces are instrumented, that is, they contain only standard rewrite steps that implement rewriting modulo an equational theory as explained in Section 3. The following definitions are auxiliary. Let t and t' be two terms. We write $t \preceq t'$ to denote that t' is an instance of t. We use the notation $t[p]$ to select the symbol rooted at position p of the term t. A substitution η is a *renaming* iff there exists a substitution η^{-1} such that, for every term t, $t\eta\eta^{-1} = t\eta^{-1}\eta = t$.

Proposition Appendix A.1 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \ldots \overset{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \to \ldots \to s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing$(\mathcal{T}, s_n^\bullet)$ is a trace slice of $\mathcal{T}$ w.r.t $s_n^\bullet$.*

Proof. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \ldots \overset{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in the conditional rewrite theory $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then,

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

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

$$\langle s_0 \to^* s_n,\ s_n^\bullet,\ \mathit{true}\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s_{n-1}^\bullet,\ B_{n-1}^\bullet\rangle \bullet\!\!\to^* \langle s_0,\ s_0^\bullet,\ B_0^\bullet\rangle \tag{A.1}$$

First, by the definition of the functions *slice-step* and *process-condition*, observe that

$$B_i^\bullet = \mathit{true} \wedge \bigwedge_{j=0}^{n-1} b_j\Phi_j \qquad \text{for all } i = 0, \ldots, n-1 \tag{A.2}$$

where, for each $j = 0, \ldots, n-1$, b_j is a conjunction of equational conditions that occur in the conditional rewrite theory $\mathcal{R}$ and Φ_j is a substitution.

Now, to prove the proposition, simply observe that

1. each $s_i^\bullet$, $i = 0, \ldots, n-1$, is built in the transition

   $$\langle s_0 \to^* s_{i+1},\ s_{i+1}^\bullet,\ B_{i+1}^\bullet\rangle \bullet\!\!\to \langle s_0 \to^* s_i,\ s_i^\bullet,\ B_i^\bullet\rangle$$

   by calling the function *slice-step* on the rewrite step $s_i \overset{r_{i+1},\sigma_{i+1},w_{i+1}}{\to} s_{i+1}$, the term slice $s_{i+1}^\bullet$, and the Boolean condition $B_{i+1}^\bullet$. Specifically, the call to *slice-step* generates a term $s_i^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$ that is either equal to $s_{i+1}^\bullet$ (See Line 4 of the function *slice-step*) or derived from $s_{i+1}^\bullet$ by replacing an appropriate subterm of $s_{i+1}^\bullet$ with an instance of the left-hand side of the rule r_{i+1} (See Line 12 of the function *slice-step*). In both cases, $s_i^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$.
2. By A.2, $B_0^\bullet$ is the conjunction of Boolean conditions $\mathit{true} \wedge b_{n-1}\Phi_{n-1} \wedge \ldots \wedge b_0\Phi_0$. Hence, $B_0^\bullet$ is a Boolean condition.

Therefore, $[s_0^\bullet \to^* s_n^\bullet,\ B_0^\bullet]$ is a trace slice of $\mathcal{T}$ w.r.t $s_n^\bullet$ by Definition 4.6. ■

Theorem 4.19 (weak correctness). *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} ... \overset{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \to ... \to s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing$(\mathcal{T}, s_n^\bullet)$ is a weakly correct trace slice of $\mathcal{T}$ w.r.t $s_n^\bullet$.*

Proof. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in the conditional rewrite theory $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then,

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

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

$$\langle s_0 \to^* s_n,\ s_n^\bullet,\ \mathit{true}\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s_{n-1}^\bullet,\ B_{n-1}^\bullet\rangle \bullet\!\!\to^* \langle s_0,\ s_0^\bullet,\ B_0^\bullet\rangle \tag{A.3}$$

By Proposition Appendix A.1, the outcome $[s_0^\bullet \to^* s_n^\bullet,\ B_0^\bullet]$ is a trace slice. Therefore, we just need to prove that $[s_0^\bullet \to^* s_n^\bullet,\ B_0^\bullet]$ is weakly correct, that is, there exists a term s_0' with $s_0^\bullet \propto^{B_0^\bullet} s_0'$, and an execution trace $s_0' \to s_1' \to \dots \to s_n'$ in $\mathcal{R}$ such that

i) $s_i' \to s_{i+1}'$ is either the rewrite step $s_i' \overset{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to} s_{i+1}'$ or $s_i' = s_{i+1}'$, $i = 0, \dots, n-1$;

ii) $s_i^\bullet$ is a term slice of s_i', for all $i = 0, \dots, n$.

The proof proceeds by induction on the total number of rewrite steps included in the execution trace $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_n,\sigma_n,w_n}{\to} s_n$ that we denote by $\mathcal{N}(\mathcal{T})$. Observe that $\mathcal{N}(\mathcal{T})$ also includes all the internal rewrites that are needed to prove the validity of the conditions involved in the conditional rewrite steps that occur in $\mathcal{T}$.

$\mathcal{N}(\mathcal{T}) = \mathbf{0}$. The execution trace $\mathcal{T}$ is empty, and hence the theorem vacuously holds.

$\mathcal{N}(\mathcal{T}) > \mathbf{0}$. We have a nonempty execution trace $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_{n-1},\sigma_{n-1},w_{n-1}}{\to} s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n$ and a slicing criterion $s_n^\bullet$ to which backward trace slicing is applied. Specifically, by applying *backward-slicing*$(\mathcal{T}, s_n^\bullet)$, the transition sequence $\mathcal{S}$ in A.3 is produced.

We consider the execution trace $\mathcal{T}^{n-1} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_{n-1},\sigma_{n-1},w_{n-1}}{\to} s_{n-1}$. Since $\mathcal{N}(\mathcal{T}^{n-1}) < \mathcal{N}(\mathcal{T})$, the inductive hypothesis holds for $\mathcal{T}^{n-1}$ and the slicing criterion $s_{n-1}^\bullet$. Thus, *backward-slicing*$(s_0 \to^* s_{n-1}, s_{n-1}^\bullet) = [s_0^\bullet \to^* s_{n-1}^\bullet,\ B_{0'}^\bullet]$ and $[s_0^\bullet \to^* s_{n-1}^\bullet,\ B_{0'}^\bullet]$ is a weakly correct trace slice of $\mathcal{T}^{n-1}$ w.r.t. the slicing criterion $s_{n-1}^\bullet$.

Then, we consider the rewrite step $s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n$. By Definition 4.11, the sliced counterpart, $s_{n-1}^\bullet \to s_n^\bullet$, of the considered rewrite step, is computed in the first transition of $\mathcal{S}$ (that is, $\langle s_0 \to^* s_n,\ s_n^\bullet,\ \mathit{true}\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s_{n-1}^\bullet,\ B_{n-1}^\bullet\rangle$) by executing *slice-step*$(s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n, s_n^\bullet, \mathit{true})$ which returns the pair $(s_{n-1}^\bullet, B_{n-1}^\bullet)$. We distinguish the two following cases.

Case $w_n \notin \mathcal{MP}os(s_n^\bullet)$**.** In this case, *slice-step*$(s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n, s_n^\bullet, true)$ yields the pair $(s_{n-1}^\bullet, B_{n-1}^\bullet)$ such that $s_{n-1}^\bullet = s_n^\bullet$ and $B_{n-1}^\bullet = true$. Thus, the trace slice for $\mathcal{T}$ w.r.t. $s_n^\bullet$ must be of the form $[s_0^\bullet \to^* s_{n-1}^\bullet = s_n^\bullet,\ B_{0'}^\bullet \wedge true]$. Since $[s_0^\bullet \to^* s_{n-1}^\bullet,\ B_{0'}^\bullet]$ is a weakly correct trace slice of $\mathcal{T}^{n-1}$ w.r.t. the slicing criterion $s_{n-1}^\bullet$, there exists a term s_0' with $s_0^\bullet \propto^{B_{0'}^\bullet} s_0'$, and an execution trace $s_0' \to s_1' \to \ldots \to s_{n-1}'$ in $\mathcal{R}$ such that

i) $s_i' \to s_{i+1}'$ is either the rewrite step $s_i' \overset{r_{i+1},\sigma_{i+1}',w_{i+1}}{\to} s_{i+1}'$ or $s_i' = s_{i+1}'$, $i = 0, \ldots, n-2$;

ii) $s_i^\bullet$ is a term slice of s_i', for all $i = 0, \ldots, n-1$.

Then, it also holds that $s_0^\bullet \propto^{B_{0'}^\bullet \wedge true} s_0'$. Furthermore, for the execution trace $s_0' \to s_1' \to \ldots \to s_{n-1}' = s_n'$ in $\mathcal{R}$, we have

i) $s_i' \to s_{i+1}'$ is either the rewrite step $s_i' \overset{r_{i+1},\sigma_{i+1}',w_{i+1}}{\to} s_{i+1}'$ or $s_i' = s_{i+1}'$, $i = 0, \ldots, n-1$;

ii) $s_i^\bullet$ is a term slice of s_i', for all $i = 0, \ldots, n$.

Therefore, $[s_0^\bullet \to^* s_{n-1}^\bullet = s_n^\bullet,\ B_{0'}^\bullet \wedge true]$ is a weakly correct trace slice of $\mathcal{T}$ w.r.t. $s_n^\bullet$.

Case $w_n \in \mathcal{MP}os(s_n^\bullet)$**.** Let r_n be te rewrite rule $\lambda_n \to \rho_n$ *if* C_n. In this case, *slice-step*$(s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n, s_n^\bullet, true)$ yields the pair $(s_{n-1}^\bullet, B_{n-1}^\bullet)$ such that $s_{n-1}^\bullet = s_n^\bullet[\lambda_n\phi]_{w_n}$, for some substitution ϕ, and $B_{n-1}^\bullet$ is the conjunction of some equational conditions needed to evaluate the condition $C_n\sigma_n$ in the rewrite step $s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n$. Thus, the trace slice for $\mathcal{T}$ w.r.t. $s_n^\bullet$ must be of the form$[s_0^\bullet \to^* s_n^\bullet[\lambda\psi]_{w_n} \to s_n^\bullet,\ B_{0'}^\bullet \wedge B_{n-1}^\bullet]$, where $B_{0'}^\bullet$ is the compatibility condition computed by slicing the execution trace $\mathcal{T}^{n-1}$.

Now, we consider the trace slice $[s_0^\bullet \to^* s_{n-1}^\bullet,\ B_{0'}^\bullet]$, which is weakly correct by inductive hypothesis. By Definition 4.7, we know that there exists a term s_0' with $s_0^\bullet \propto^{B_{0'}^\bullet} s_0'$, and an execution trace $s_0' \to s_1' \to \ldots \to s_{n-1}'$ in $\mathcal{R}$ such that

i) $s_i' \to s_{i+1}'$ is either the rewrite step $s_i' \overset{r_{i+1},\sigma_{i+1}',w_{i+1}}{\to} s_{i+1}'$ or $s_i' = s_{i+1}'$, $i = 0, \ldots, n-2$;

ii) $s_i^\bullet$ is a term slice of s_i', for all $i = 0, \ldots, n-1$.

In particular, we can choose $s_0' = s_0$. This way, all the descendants of s_0 will be kept in the execution trace $s_0 = s_0' \to s_1' \to \ldots \to s_{n-1}'$. Hence, $s'_{n-1|w_n} = s_{n-1|w_n} = \lambda_n\sigma_n$. Furthermore, $s_0^\bullet \propto^{B_{0'}^\bullet \wedge B_{n-1}^\bullet} s_0 = s_0'$. Thus, by definition of rewriting, $s_{n-1}' \overset{r_n,\sigma_n',w_n}{\to} s_n'$ with $s'_{n|w_n} = \rho_n\sigma_n' = \rho_n\sigma_n = s_{n|w_n}$. Also, $s_n^\bullet$ is a term slice of s_n', since $s_n^\bullet$ is a slicing criterion for $\mathcal{T}$.

Therefore, $[s_0^\bullet \to^* s_n^\bullet[\lambda\psi]_{w_n} \to s_n^\bullet,\ B_{0'}^\bullet \wedge B_{n-1}^\bullet]$ is a trace slice for $\mathcal{T}$ w.r.t. $s_n^\bullet$.

■

The next lemmata state some useful properties on the outcome of the *slice-step* function, that are necessary to prove the strong correctness of backward trace slicing.

Lemma Appendix A.2 *Let $r = \lambda \to \rho$ if C be a rewrite rule of a conditional rewrite theory $\mathcal{R}$ that satisfies the separation property and only contains equational conditions. Let $\mathcal{T} : s \overset{r,\sigma,w}{\to} t$ be a (one-step) execution trace. Let $t^\bullet$ be a term slice of t and $B^\bullet_{prev}$ be a Boolean condition. Then, slice-step$(\mathcal{T}, t^\bullet, B^\bullet_{prev})$ computes the pair $(s^\bullet, B^\bullet)$ such that, for each term s', with $s^\bullet \propto^{B^\bullet} s'$,*

i) $s^\bullet$ is a term slice of s';

ii) there exists a term t' such that $s' \overset{r,\sigma',w}{\to} t'$ or $s' = t'$, and $t^\bullet$ is a term slice of t'.

Proof. Let $\mathcal{R}$ be a conditional rewrite theory that satisfies the separation property and only contains equational conditions. Let $r = \lambda \to \rho$ *if* C be a rewrite rule in $\mathcal{R}$, hence C is a conjunction of equational conditions $c_1 \wedge \ldots \wedge c_m$. Let $\mathcal{T} : s \overset{r,\sigma,w}{\to} t$ be a (one-step) execution trace. Let $t^\bullet$ be a term slice of t and $B^\bullet_{prev}$ be a Boolean condition. Let $(s^\bullet, B^\bullet)$ be the outcome of *slice-step*$(\mathcal{T}, t^\bullet, B^\bullet_{prev})$. We prove *i*) and *ii*) separately.

i) The proof that $s^\bullet$ is a term slice of s' directly derives from the hypothesis of the lemma. In fact, since $s^\bullet \propto^{B^\bullet} s'$, by Definition 4.4, there exists a substitution η such that $s^\bullet\eta = s'$. Hence, s' is an instance of the term $s^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$, which implies that $s^\bullet$ is a term slice of s'.

ii) We distinguish two cases: $w \notin \mathcal{MP}os(t^\bullet)$ and $w \in \mathcal{MP}os(t^\bullet)$.

Case $w \notin \mathcal{MP}os(t^\bullet)$**.** As w is not a meaningful position of $t^\bullet$, the execution of *slice-step*$(s \overset{r,\sigma,w}{\to} t, t^\bullet, B^\bullet_{prev})$ generates the pair $(s^\bullet, B^\bullet)$ with $s^\bullet = t^\bullet$ and $B^\bullet = B^\bullet_{prev}$ (see Lines 1–4 of the *slice-step* algorithm in Figure 3), which indicates that the rewrite step $s \overset{r,\sigma,w}{\to} t$ does not contribute to producing the meaningful symbols of $t^\bullet$.

Now, given an arbitrary term s', such that $s^\bullet \propto^{B^\bullet} s'$, we have that $s^\bullet \preceq s'$. Hence, $s' = s^\bullet\eta$ for some substitution η. Since $s^\bullet = t^\bullet$, the following equivalences hold: $s' = s^\bullet\eta = t^\bullet\eta = t'$. Therefore, $s' = t'$. Furthermore, $t^\bullet$ is a term slice of t', because $s' = t'$, $s^\bullet = t^\bullet$ and $s^\bullet$ is a term slice of s'.

Case $w \in \mathcal{MP}os(t^\bullet)$**.** In this case, the *else* branch of the function *slice-step* is executed, that is, Lines 6–13. First, note that the execution of Lines 6–8 generates a substitution ψ_ρ such that

$$\rho^\bullet\psi_\rho = t^\bullet_w\eta \tag{A.4}$$

for some renaming η, and the term slice $\rho^\bullet = slice(\rho, \overline{\mathcal{MP}os}(\mathcal{V}ar^\bullet(B^\bullet_{prev}), t^\bullet_{|w}) \cap \mathcal{P}os(\rho))$ of ρ. Basically, Equation A.4 holds, since

1. Line 6 yields a substitution θ which binds each variable in ρ to a new, freshly generated $\bullet$-variable.
2. By Definition 4.2, Line 7 computes a term slice $\rho^\bullet$ of ρ which includes all and only the symbols in ρ that are meaningful symbols (or $\bullet$-variables that appear in $B^\bullet_{prev}$) of $t^\bullet_{|w}$;
3. Line 8 computes the substitution $\mu = match_{\rho^\bullet}(t^\bullet_{|w})$ such that $\rho^\bullet\mu = t^\bullet_{|w}$, and —by Definition 4.13— $\psi_\rho = \langle\!|\theta, \mu|\!\rangle$ updates θ with the meaningful information of $t^\bullet_{|w}$ encoded in μ. Hence, for each position $p \in \mathcal{P}os(\rho^\bullet\psi_\rho)$, we have either $\rho^\bullet\psi_\rho[p] = t^\bullet_{|w}[p]$ and $p \in \mathcal{MP}os(t^\bullet_{|w})$ or $\rho^\bullet\psi_\rho[p] = \bullet_i \neq \bullet_j = t^\bullet_{|w}[p]$ for some $\bullet_i, \bullet_j \in \mathcal{V}^\bullet$. Thus, by choosing the renaming $\eta = \{\bullet_i/\bullet_j \mid \rho^\bullet_{|p} = \bullet_i, t^\bullet_{|w.p} = \bullet_j, p \in \mathcal{P}os(\rho^\bullet)\}$, we get $\rho^\bullet\psi_\rho = t^\bullet_{|w}\eta$.

Now, Lines 9–11 evaluate the rule condition $C = c_1, \wedge \ldots \wedge c_n$ by invoking the function *process-condition*. Specifically, for each c_i, *process-condition* generates the refinement $\langle\!|\psi_\rho, \psi_n \ldots \psi_{i+1}|\!\rangle$ of ψ_ρ. Since each c_i is an equational condition, at the end of the **for** loop, we obtain the refinement $\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle$ such that each ψ_i, $i = 1, \ldots, n$, is the identity substitution $\{\}$. Hence, for each term t,

$$t\psi_\rho = t\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle \tag{A.5}$$

Line 12 constructs the term $s^\bullet = (t^\bullet[\lambda\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle]_w)$, and Line 13 builds the compatibility condition $B^\bullet$. Note that $B^\bullet$ contains a sliced counterpart of every equational condition c_i in $c_1 \wedge \ldots \wedge c_m$.

Now, we consider an arbitrary term s', such that $s^\bullet \propto^{B^\bullet} s'$. Since $s^\bullet \propto^{B^\bullet} s'$, it also holds that $s^\bullet \preceq s'$. Hence, there exists a substitution ψ such that $s' = (t^\bullet[\lambda\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle]_w)\psi = t^\bullet\psi[\lambda\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi]_w$. Thus, we can rewrite s' using the rule r at position w,

$$s' = t^\bullet\psi[\lambda\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi]_w \overset{r,\langle\!|\psi_\rho,\psi_n\ldots\psi_1|\!\rangle\psi,w}{\rightarrow} t^\bullet\psi[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi]_w.$$

Let t' be $t^\bullet\psi[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi]_w$. Then, $t' = t^\bullet\psi[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi]_w = (t^\bullet[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle]_w)\psi$. Hence, $t^\bullet \preceq t'$ holds as well. Indeed, we have

$$\begin{aligned}
t' = (t^\bullet[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle]_w)\psi &\succeq (t^\bullet[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle]_w) \text{ (by definition of instance)} \\
&= t^\bullet[\rho\psi_\rho]_w \text{ (by A.5)} \\
&\succeq t^\bullet[\rho^\bullet\psi_\rho]_w \text{ (as } \rho^\bullet \preceq \rho \text{ by definition of term slice)} \\
&= t^\bullet[t^\bullet\eta]_w \text{ (by Equation A.4)} \\
&\succeq t^\bullet \text{ (by definition of instance)}
\end{aligned}$$

Finally, observe that, by choosing

$$\sigma' = \langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi \text{ and } t' = t^\bullet\psi[\rho\langle\!|\psi_\rho, \psi_n \ldots \psi_1|\!\rangle\psi]_w$$

we have that $s' \overset{r,\sigma',w}{\to} t'$ and $t^\bullet$ is a term slice of t'. Hence, the lemma holds even in the case when $w \notin \mathcal{MP}os(t^\bullet)$.

■

Lemma Appendix A.3 *Let $r = \lambda \to \rho$ if C be a rewrite rule of a conditional rewrite theory $\mathcal{R}$ that satisfies the separation property and only contains matching conditions or rewrite expressions. Let $\mathcal{T} : s \overset{r,\sigma,w}{\to} t$ be a (one-step) execution trace. Let $t^\bullet$ be a term slice of t and $B^\bullet_{prev}$ be a Boolean condition. Then, slice-step$(\mathcal{T}, t^\bullet, B^\bullet_{prev})$ computes the pair $(s^\bullet, B^\bullet)$ such that, for each term s', with $s^\bullet \propto^{B^\bullet} s'$,*

i) $s^\bullet$ is a term slice of s';

ii) there exists a term t' such that $s' \overset{r,\sigma',w}{\to} t'$ or $s' = t'$, and $t^\bullet$ is a term slice of t'.

Proof. The proof is analogous to proof of Lemma Appendix A.2. Basically, the main difference lies in the evaluation of the function *process-condition* (Lines 9–11 of the *slice-step* function). In this case, each matching condition/rewrite expression c_i is directly encoded in the corresponding computed substitution refinement, while the compatibility condition is left unchanged. More specifically, for each c_i, *process-condition* generates the refinement $\langle\!\langle \psi_\rho, \psi_n \dots \psi_{i+1} \rangle\!\rangle$ of ψ_ρ. Hence, at the end of the **for** loop, we obtain the Boolean compatibility condition $B^\bullet = true$, and the refinement $\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle$ such that, for each term t,

$$t\psi_\rho \preceq t\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle. \tag{A.6}$$

Therefore, by considering an arbitrary term s', such that $s^\bullet \propto^{true} s'$, we have that $s^\bullet \preceq s'$. Hence, there exists a substitution ψ such that $s' = (t^\bullet[\lambda\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle]_w)\psi = t^\bullet\psi[\lambda\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi]_w$. Thus, we can rewrite s' using the rule r at position w,

$$s' = t^\bullet\psi[\lambda\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi]_w \overset{r,\langle\!\langle \psi_\rho,\psi_n \dots \psi_1 \rangle\!\rangle\psi,w}{\to} t^\bullet\psi[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi]_w.$$

Let t' be $t^\bullet\psi[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi]_w$. Then, $t' = t^\bullet\psi[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi]_w = (t^\bullet[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle]_w)\psi$. Thus, $t^\bullet \preceq t'$ holds as well. Indeed, we have

$$\begin{aligned} t' = (t^\bullet[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle]_w)\psi &\succeq (t^\bullet[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle]_w) \text{ (by definition of instance)} \\ &\succeq t^\bullet[\rho\psi_\rho]_w \text{ (by A.6)} \\ &\succeq t^\bullet[\rho^\bullet\psi_\rho]_w \text{ (as } \rho^\bullet \preceq \rho \text{ by definition of term slice)} \\ &= t^\bullet[t^\bullet\eta]_w \text{ (by Equation A.4)} \\ &\succeq t^\bullet \text{ (by definition of instance)} \end{aligned}$$

Finally, observe that, by choosing

$$\sigma' = \langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi \text{ and } t' = t^\bullet\psi[\rho\langle\!\langle \psi_\rho, \psi_n \dots \psi_1 \rangle\!\rangle\psi]_w$$

we have that $s' \overset{r,\sigma',w}{\to} t'$ and $t^\bullet$ is a term slice of t'. ■

Theorem 4.22 (strong correctness). *Let $\mathcal{R}$ be a conditional rewrite theory that satisfies the separation property. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} ... \overset{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \to ... \to s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing$(\mathcal{T}, s_n^\bullet)$ is a strongly correct trace slice of $\mathcal{T}$ w.r.t $s_n^\bullet$.*

Proof. Let us consider a conditional rewrite theory $\mathcal{R}$ that satisfies the separation property and only contains equational conditions. The case when $\mathcal{R}$ is a conditional rewrite theory that satisfies the separation property and only includes matching conditions or rewrite expressions is perfectly similar and requires the application of Lemma Appendix A.3 instead of using Lemma Appendix A.2.

Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_n,\sigma_n,w_n}{\to} s_n$ be an execution trace in $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then,

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

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

$$\langle s_0 \to^* s_n,\ s_n^\bullet,\ \mathit{true}\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s_{n-1}^\bullet,\ B_{n-1}^\bullet\rangle \bullet\!\!\to^* \langle s_0,\ s_0^\bullet,\ B_0^\bullet\rangle \tag{A.7}$$

By Proposition Appendix A.1, the outcome $[s_0^\bullet \to^* s_n^\bullet,\ B_0^\bullet]$ is a trace slice. Therefore, we just need to prove that $[s_0^\bullet \to^* s_n^\bullet,\ B_0^\bullet]$ is strongly correct, that is, for every term s_0' such that $s_0^\bullet \propto^{B_0^\bullet} s_0'$, there exists an execution trace $s_0' \to s_1' \to \dots \to s_n'$ in $\mathcal{R}$ such that

i) $s_i' \to s_{i+1}'$ is either the rewrite step $s_i' \overset{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to} s_{i+1}'$ or $s_i' = s_{i+1}'$, $i = 0, \dots, n-1$;

ii) $s_i^\bullet$ is a term slice of s_i', for all $i = 0, \dots, n$.

We proceed by induction on the total number of rewrite steps included in the execution trace $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_n,\sigma_n,w_n}{\to} s_n$ that we denote by $\mathcal{N}(\mathcal{T})$. Observe that $\mathcal{N}(\mathcal{T})$ also includes all the internal rewrites that are needed to prove the validity of the conditions involved in the conditional rewrite steps that occur in $\mathcal{T}$.

$\mathcal{N}(\mathcal{T}) = \mathbf{0}$. In this case the execution trace $\mathcal{T}$ is empty, and hence the result vacuously holds.

$\mathcal{N}(\mathcal{T}) > \mathbf{0}$. We have a nonempty execution trace $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_{n-1},\sigma_{n-1},w_{n-1}}{\to} s_{n-1} \overset{r_n,\sigma_n,w_n}{\to} s_n$ and a slicing criterion $s_n^\bullet$ to which backward trace slicing is applied. Specifically, by applying *backward-slicing*$(\mathcal{T}, s_n^\bullet)$, the transition sequence $\mathcal{S}$ in A.7 is produced.

Now, we consider the execution trace $\mathcal{T}^{n-1} : s_0 \overset{r_1,\sigma_1,w_1}{\to} \dots \overset{r_{n-1},\sigma_{n-1},w_{n-1}}{\to} s_{n-1}$. Since $\mathcal{N}(\mathcal{T}^{n-1}) < \mathcal{N}(\mathcal{T})$, the inductive hypothesis holds for $\mathcal{T}^{n-1}$ and the slicing criterion $s_{n-1}^\bullet$. Hence, we have that *backward-slicing*$(s_0 \to^* s_{n-1}, s_{n-1}^\bullet) = [s_0^\bullet \to^* s_{n-1}^\bullet,\ B_{0'}^\bullet]$ and $[s_0^\bullet \to^* s_{n-1}^\bullet,\ B_{0'}^\bullet]$ is a trace slice of $\mathcal{T}^{n-1}$ w.r.t. the slicing criterion $s_{n-1}^\bullet$.

Therefore, there exists the transition sequence in $(Conf, \bullet\!\!\rightarrow)$

$$\langle s_0 \rightarrow^* s_{n-1},\ s^\bullet_{n-1},\ true\rangle \bullet\!\!\rightarrow^* \langle s_0,\ s^\bullet_0,\ B^\bullet_{0'}\rangle$$

with $B^\bullet_{0'} = true \wedge b_{n-2}\Phi_{n-2} \wedge \ldots \wedge b_0\Phi_0$, and the following properties hold: for every term s'_0 such that $s^\bullet_0 \propto^{B^\bullet_{0'}} s'_0$, there exists an execution trace $s'_0 \rightarrow s'_1 \rightarrow \ldots \rightarrow s'_{n-1}$ 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 $s'_i = s'_{i+1}$, $i = 0, \ldots, n-2$;

ii) $s^\bullet_i$ is a term slice of s'_i, for all $i = 1, \ldots, n-1$.

Furthermore, by Definition 4.11, the first transition of $\mathcal{S}$ (i.e., $\langle s_0 \rightarrow^* s_n,\ s^\bullet_n,\ true\rangle \bullet\!\!\rightarrow \langle s_0 \rightarrow^* s_{n-1},\ s^\bullet_{n-1},\ B^\bullet_{n-1}\rangle$) is generated by executing $\mathit{slice\text{-}step}(s_{n-1} \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n, s^\bullet_n, true)$ which returns the pair $(s^\bullet_{n-1}, B^\bullet_{n-1})$. By Lemma Appendix A.2, we have that for every term s'_{n-1} such that $s^\bullet_{n-1} \propto^{B^\bullet_{n-1}} s'_{n-1}$, there exists $s'_{n-1} \rightarrow s'_n$ in $\mathcal{R}$ such that

i) $s^\bullet_{n-1}$ is a term slice of s'_{n-1};

ii) $s'_{n-1} \rightarrow s'_n$ is either the rewrite step $s'_{n-1} \overset{r_{i+1},\sigma'_{i+1},w_{i+1}}{\rightarrow} s'_n$ or $s'_{n-1} = s'_n$ and $s^\bullet_n$ is a term slice of s'_n;

Now observe that $B^\bullet_0 = B^\bullet_{0'} \wedge B^\bullet_{n-1}$ which implies that both $B^\bullet_{0'}$ and $B^\bullet_{n-1}$ are Boolean conditions which are more restrictive than $B^\bullet_0$. Consequently, the transition sequence $\mathcal{S}$ can be reformulated as follows

$$\langle s_0 \rightarrow^* s_n,\ s^\bullet_n,\ true\rangle \bullet\!\!\rightarrow \langle s_0 \rightarrow^* s_{n-1},\ s^\bullet_{n-1},\ B^\bullet_{n-1}\rangle \bullet\!\!\rightarrow^* \langle s_0,\ s^\bullet_0, B^\bullet_{n-1} \wedge B^\bullet_{0'}\rangle$$

Thus, it also holds that, for every term s'_0 such that $s^\bullet_0 \propto^{B^\bullet_0} s'_0$, 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) $s^\bullet_i$ is a term slice of s'_i, for all $i = 1, \ldots, n$.

■