Alpuente Frasnedo, M.; Ballis, D.; Frechina Navarro, F.; Sapina, J. (2015). Exploring Conditional Rewriting Logic Computations. Journal of Symbolic Computation. 69:3-39. doi:10.1016/j.jsc.2014.09.028.

Exploring Conditional Rewriting Logic Computations ⋆

M. Alpuente [a] D. Ballis [b] F. Frechina [a] J. Sapiña [a]

[a] *DSIC-ELP, Universitat Politècnica de València, Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain,*

[b] *CLIP-Lab, Technical University of Madrid, E-28660, Boadilla del Monte, Madrid, Spain, and DIMI, Università degli Studi di Udine, Via delle Scienze 206, 33100, Udine, Italy.*

Key Words: rewriting logic, trace exploration, maude, conditional rewrite theories

Abstract

Trace exploration is concerned with techniques that allow computation traces to be dynamically searched for specific contents. Depending on whether the exploration is carried backward or forward, trace exploration techniques allow *provenance tracking* or *impact tracking* to be done. The aim of *provenance tracking* is to show how (parts of) a program output depends on (parts of) its input and to help estimate which input data need to be modified to accomplish a change in the outcome. The aim of *impact tracking* is to identify the scope and potential consequences of changing the program input. Rewriting Logic (RWL) is a logic of change that supplements (an extension of) the equational logic by adding rewrite rules that are used to describe (nondeterministic) transitions between states. In this paper, we present a rich and highly dynamic, parameterized technique for the *forward* inspection of RWL computations that allows the nondeterministic execution of a given *conditional* rewrite theory to be followed up in different ways. With this technique, an analyst can browse, slice, filter, or search the traces as they come to life during the program execution. The navigation of the trace is driven by a user-defined, inspection criterion that specifies the required exploration mode. By selecting different inspection criteria, one can automatically derive a family of practical algorithms such as program steppers and more sophisticated dynamic trace slicers that compute summaries of the computation tree, thereby facilitating the dynamic detection of control and data dependencies across the tree. Our methodology, which is implemented in the Anima graphical tool, allows users to evaluate the effects of a given statement or instruction in isolation, track input change impact, and gain insight into program behavior (or misbehavior).

1. Introduction

Dynamic analysis is crucial for understanding the behavior of large systems. Dynamic information is typically represented using execution traces whose analysis is almost impracticable without adequate tool support. Existing tools for analyzing large execution traces commonly rely on a set of visualization techniques that facilitate the exploration of the trace content. Common capabilities of these tools include the option to simplify the traces by hiding some specific contents and stepping the program execution while searching for particular components. Nearly all modern IDEs, debuggers, and testing tools currently support this mode of execution optionally, where animation is typically achieved either by forcing execution breakpoints, instruction simulation, or code instrumentation.

Rewriting Logic is a very general *logical* and *semantic framework* that is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems (Baggi et al., 2009) and Web systems (Alpuente et al., 2006, 2009, 2010b, 2013a, 2014c)). Rewriting Logic is efficiently implemented in the high-performance system Maude (Clavel et al., 2011). Roughly speaking, a *rewriting logic theory* seamlessly combines a *term rewriting system* (TRS) with an *equational theory* that may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are performed *modulo* the equations and axioms. In recent years, debugging and optimization techniques based on RWL have received growing attention (Alpuente et al., 2010a, 2011; Martí-Oliet and Meseguer, 2002; Riesco et al., 2009, 2010), but to the best of our knowledge, no trace inspection tool or skilled program animator for conditional RWL theories has been formally developed to date.

To debug Maude programs, Maude has a basic tracing facility that allows the user to advance through the program execution stepwisely with the possibility to set break points and lets him/her select the statements to be traced, except for the application of algebraic axioms that are not under user control and are never recorded explicitly in the trace. All rewrite steps that are obtained by applying the equations or rules for the selected function symbols are shown in the output trace so that the only way to simplify the displayed view of the trace is by manually fixing the traceable equations or rules. Thus, the trace is typically 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 only being amenable for manual inspection by the user. This is in contrast with the enriched traces described in this work, which are complete (all execution steps are recorded by default) and can be sliced automatically so that they can be dramatically simplified in order to facilitate a specific analysis. Also, the trace can be

⋆ This work has been partially supported by the EU (FEDER) and the Spanish MEC project ref. TIN2010-21062-C02-02, the Spanish MICINN complementary action ref. TIN2009-07495-E, 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 J. Sapiña was supported by FPI-UPV grant SP2013-0083.

Email addresses: alpuente@dsic.upv.es (M. Alpuente), demis.ballis@uniud.it (D. Ballis), ffrechina@dsic.upv.es (F. Frechina), jsapina@dsic.upv.es (J. Sapiña).

directly displayed or delivered in its meta-level representation, which is very useful for further automated manipulation.

Contributions. This paper presents the first semantic-based, parametric trace exploration technique for RWL computations in conditional rewrite theories that involve rewriting modulo associativity (A), commutativity (C), and unity (U) axioms. Our technique is based on a generic animation algorithm that can be tuned to work with different modalities, including *incremental stepping* and *automated forward slicing*, which drastically reduce the size and complexity of the traces under examination. The algorithm is fully general and can be applied for debugging as well as for optimizing RWL-based tools that manipulate conditional RWL theories. Our formulation takes into account the precise way in which Maude mechanizes the conditional rewriting process modulo equational theories that may contain any combination of associativity, commutativity, and unity axioms for different binary operators, and it revisits all those rewrite steps in an informed, fine-grained way where each small step corresponds to the application of an equational axiom, conditional equation, or conditional rule. This allows us to inspect the execution trace with regard to the set of symbols of interest (input symbols) by tracing them along the execution trace so that, in the case of the forward slicing modality, all data that are not descendants of the observed symbols are filtered out. The ideas are implemented and tested in a graphical tool called Anima, which provides a mighty and versatile environment for the dynamic analysis of RWL computations.

Related Work. Dependency analysis techniques provide a formal foundation for forms of provenance that are intended to highlight parts of the program input on which a part of its output depends (Cheney et al., 2011). This is essentially achieved by computing the origins of the given results and has proved to be useful in many contexts such as efficient memorization and caching (Abadi et al., 1996), aiding program debugging via slicing (Alpuente et al., 2013c, 2014a; Field and Tip, 1994), information-flow security (Sabelfeld and Myers, 2003) and several other forms of program analysis techniques, just to mention a few. A great deal of work has been done on each of these topics, and we refer to (Cheney et al., 2011) for further references that we cannot survey here.

The notion of descendants (Klop, 1990) or residuals [1] (Huet and Lévy, 1970; O'Donnell, 1977) with its inverse notion of ancestors or origins is classical in the theory of rewriting, both in first-order term rewriting and in higher-order rewriting, such as lambda calculus. While dependency provenance provides information about the origins of (or influences upon) a given result, the notion of descendants is the key for impact evaluation, that is, to assess the changes that can be attributed to a particular input or intervention (Bethke et al., 2000). For orthogonal term rewriting systems (i.e., left–linear and overlap–free), a refined version of the descendant/ancestor relation, called origin tracking, was first introduced in (Klop, 1990). Several variants of this notion have been studied, sometimes with applications that are similar to the ones described in this paper (see (Bethke et al., 2000) for references). An extension of (Klop, 1990) for all TRSs is described in (TeReSe, 2003). A method for implementing origin tracking in conditional term rewriting systems is given in (Van Deursen et al., 1993).

For the rich framework of conditional rewrite theories, an incremental, *backward* conditional trace slicer is presented in (Alpuente et al., 2012a,b, 2014a). This framework

[1] In the literature, the term 'residual' is usually reserved for a descendant of a redex (Bethke et al., 2000).

generates a trace slice of an execution trace $\mathcal{T}$ by tracing back the origins of the set of symbols of interest along (an instrumented version of) $\mathcal{T}$. This can be very helpful in debugging since any information that is not strictly needed to deliver a critical part of the result is disregarded. However, for the dual problem of "which parts of the output might be impacted by a specific component of the input", forward expansion is needed. To date, forward expansion has been mostly overlooked in RWL research.

Program animators have existed since the early years of programming. Although several steppers have been implemented in the functional programming community (see (Clements et al., 2001) for references), none of these systems applies to the animation and dynamic forward slicing of Maude computations. An algebraic stepper for Scheme is defined and formally proved in (Clements et al., 2001), which is included in the DrScheme programming environment. In order to discover all of the steps that occur during the program evaluation, the stepper rewrites (or "instruments") the code. This is in contrast to our technique, which does not rely on program instrumentation.

A generic, static technique to infer (forward) program slices is defined in (Riesco et al., 2013). This technique relies on the formal executable semantics of the language of interest, which is given as a RWL theory. Informally, this kind of slices represent program fragments that are affected by a particular program statement with respect to a set of variables of interest. Different from our technique, this technique is static and defines program slices by using (meta-level) over-approximation.

Plan of the paper. After some preliminaries in Section 2 that recall basic notions of RWL and summarize the conditional rewriting modulo theories defined in Maude, Section 3 presents a convenient trace instrumentation technique that facilitates the stepwise inspection of Maude computations. In Section 4, we formalize trace inspection as a semantics-based forward procedure that is parameterized by the criterion for the inspection. Section 5 presents three different exploration techniques that are mechanically obtained as an instance of the generic scheme: 1) an interactive program stepper that allows rewriting logic theories to be stepwisely animated; 2) a partial stepper that is able to work with partial inputs; and 3) an automated, forward slicing technique that computes trace summaries by employing *unification* and that filters out the irrelevant data that do not derive from some selected terms of interest. Correctness results are provided for all the considered exploration techniques. The Anima tool is described in Section 6, where we discuss its suitability for the analysis of complex, textually-large system computations. Section 7 concludes.

A preliminary version of the forward inspection methodology developed in this article first appeared in (Alpuente et al., 2013b) and in its superseded version (Alpuente et al., 2014b). The current approach greatly extends the exploration methodologies of (Alpuente et al., 2013b) and (Alpuente et al., 2014b) in a number of ways. The most salient novelties are listed below.

- The current trace exploration technique applies to *conditional* rewrite theories while (Alpuente et al., 2013b) and (Alpuente et al., 2014b) can only deal with the much simpler unconditional rewrite theories.
- In this article, correctness is formally proven for all the exploration modalities, whereas (Alpuente et al., 2013b) and (Alpuente et al., 2014b) do not provide any formal result for the behavior of the unconditional forward inspection algorithms;
- The forward trace slicing technique in this article relies on a unification mechanism to propagate relevant information across the trace slices. This is in contrast with the previous approach that used a much more involved substitution refinement technique.

- Several new features have been included in the Anima system. For instance, the backward trace slicer *i*Julienne (Alpuente et al., 2014a) has been integrated into Anima enabling the possibility to explore computations both back and forth in order to validate input data or to locate programming mistakes; the computation space can be also visualized now as a graph where common subexpressions (e.g., repeated states) are shared; the tool is now equipped with an online manual; and the (sliced) computation traces used for the evaluation of the conditions of the rules/equations can be also inspected now.

  Also, the main algorithm has been optimized by porting some of its components to the C++ language and by reimplementing several Maude system modules containing non-deterministic functions by means of Maude functional modules containing much more efficient, deterministic functions. This has greatly improved performance allowing more complex computations to be quickly analyzed.

2. Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting (TeReSe, 2003) and Rewriting Logic (Meseguer, 1992). Some familiarity with the Maude language (Clavel et al., 2011, 2007) is also required.

Maude is a rewriting logic (Meseguer, 1992) specification and verification system whose operational engine is mainly based on a very efficient implementation of rewriting. Maude's basic programming statements are equations and rules. Equations are used to express deterministic computations that lead to a unique final result, while rules naturally express concurrent, nondeterministic, and possibly nonterminating computations. A Maude program containing only equations (together with the syntax declaration for sorts, operators, and variables) is called a functional module and essentially defines one or more functions by means of equations. A Maude program containing rules and possibly equations is called a system module, where the rules define transitions in a possibly concurrent system. Maude notation will be introduced "on the fly" as required.

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 (Clavel et al., 2011). 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 Σ and $(S, <)$, called *preregularity* (Clavel et al., 2011), ensures that each (well-formed) term t always has 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 can arise.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. 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 two positions w_1 and w_2, we say that w_1 and w_2 are not comparable iff $w_1 \not\leq w_2$ and $w_2 \not\leq w_1$.

Given a set of positions P, and a position $p \in P$, we say that p is *minimal* w.r.t. P, iff there does not exist a position $p' \in P$ such that $p' \leq p$ and $p' \neq p$ (i.e., p' is strictly smaller than p). 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* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $\tau(\Sigma, \mathcal{V})$, which is equal to the identity almost everywhere except over a set of variables $\{x_1, \ldots, x_n\}$. The *domain* of σ is the set $Dom(\sigma) = \{x \in \mathcal{V} \mid x\sigma \neq x\}$. By ε, we denote the *identity* substitution. The application of a substitution σ to a term t, denoted $t\sigma$, is defined by induction on the structure of terms (Baader and Snyder, 2001):

$$t\sigma = \begin{cases} x\sigma & \text{if } t = x, x \in \mathcal{V} \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n), n \geq 0 \end{cases}$$

Given two terms s and t, a substitution σ is the *matcher* of t in s, if $s\sigma = t$. The term t is an *instance* of the term s (in symbols, $s \leq t$), 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. Given two substitutions θ and θ', their *composition* $\theta\theta'$ is defined as $t(\theta\theta') = (t\theta)\theta'$ for every term t. We recall that composition is associative. A substitution σ is more general than θ, denoted by $\sigma \leq \theta$, if $\theta = \sigma\gamma$ for some substitution γ. We say that a substitution σ is a *unifier* of two terms t and t' if $t\sigma = t'\sigma$. We let $mgu(t, t')$ denote a *most general unifier* σ of t and t' (i.e., $\sigma \leq \theta$ for any other unifier θ of t and 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., $x\sigma_{\restriction V} = x\sigma$ if $x \in V$, otherwise $x\sigma_{\restriction V} = x$). Given a binary relation $\rightsquigarrow$, we define the usual *transitive* (resp., *transitive and reflexive*) closure of $\rightsquigarrow$ by $\rightsquigarrow^+$ (resp., $\rightsquigarrow^*$).

2.2. Program Equations and Rules

Our techniques in this article deal with conditional RWL theories. We consider three different kinds of conditions that may appear in a conditional Maude theory: an *equational condition* [2] e is any (ordinary) equation $t = t'$, with $t, t' \in \tau(\Sigma, \mathcal{V})$; a *matching condition* is a pair $p := t$ with $p, t \in \tau(\Sigma, \mathcal{V})$; a *rewrite expression* is a pair $t \Rightarrow p$, with $p, t \in \tau(\Sigma, \mathcal{V})$.

A labelled *conditional* equation (Maude keyword `ceq`), or simply (*conditional*) equation, is an expression of the form $[l] : \lambda = \rho$ *if* C, where l is a label (i.e., a name that identifies the equation), $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$ (with $ls(\lambda) \equiv_< ls(\rho)$), 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 $[l] : \lambda = \rho$ and use the keyword `eq` to declare it in Maude. A conditional equation $[l] : \lambda = \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible*, iff (i) $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{i=1}^{n} \mathcal{V}ar(c_i)$, and (ii) for each c_i, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is an equational condition, and $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is a matching condition $p := e$.

A labelled *conditional* rewrite rule (Maude keyword `crl`), or simply (*conditional*) rule, is an expression of the form $[l] : \lambda \Rightarrow \rho$ *if* C, where l is a label, $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$ (with

[2] 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.

$ls(\lambda) \equiv_< ls(\rho)$), 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. Unlike matching conditions, which can only use equations to evaluate the input term t, rewrite expressions can apply both equations and rewrite rules for the evaluation. When the condition C is empty, we simply write $[l] : \lambda \Rightarrow \rho$ and use the keyword `rl` to declare it in Maude. A conditional rule $[l] : \lambda \Rightarrow \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible* iff it fulfills 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 $e \Rightarrow p$, $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$.

When no confusion can arise, rule and equation labels $[l]$ are often omitted. The term λ (resp., ρ) is called *left-hand side* (resp. *right-hand side*) of the rule $\lambda \Rightarrow \rho$ *if* C (resp. equation $\lambda = \rho$ *if* C).

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.

Matching conditions and rewrite expressions are useful for performing a search through a structure without having to explicitly define a search function. This is because substitutions for matching p against $t\sigma$ need not be unique since some operators may be matched modulo equational attributes. For instance, considering that list concatenation obeys associativity with unity element `nil`, we can define two Maude equations to determine whether an element `E` occurs in a list `L` as follows:

```
ceq E in L = true if L1 E L2 := L .
 eq E in L = false [owise] .
```

where the `owise` attribute allows the second equation to be applied whenever the first equation is not applicable.

2.3. Conditional Rewrite Theories

Roughly speaking, a (conditional) rewrite theory (Meseguer, 1992) seamlessly combines a set of conditional rewrite rules (or conditional term rewriting system, CTRS), with an equational theory (also possibly conditional) that may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are applied modulo the equations and axioms. Within this framework, the system states are typically represented as elements of an algebraic data type that is specified by the equational theory, while the system computations are modeled via the rewrite rules, which describe transitions between states.

More formally, 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 $\tau(\Sigma, \mathcal{V})$, which is denoted by $=_E$.

```
mod MAZE is
  pr NAT .

  sorts Pos List State Player .
  subsort Pos < List .

  op p1 : -> Player [ctor] .
  op p2 : -> Player [ctor] .
  op nil : -> List [ctor] .
  op size : -> Nat .
  op wall : -> List .
  op exit : -> List .
  op empty : -> State [ctor] .
  op next : List Nat -> Pos .
  op isOk : List -> Bool .
  op _in_ : Pos List -> Bool .
  op `{_,_,_`} : Player List Nat -> State [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op <_,_> : Nat Nat -> Pos [ctor] .
  op _||_ : State State -> State [ctor assoc comm id: empty] .

  vars X Y N M M1 M2 : Nat .
  vars P Q : Pos .
  vars L L1 L2 : List .
  vars PY PY1 PY2 : Player .

  eq [s] : size = 5 . --- Assumption: 5x5 maze
  eq [wl] : wall = < 1,3 > < 1,5 > < 2,1 > < 2,4 > < 2,5 > < 3,3 > < 3,4 >
                   < 4,2 > < 4,3 > < 5,4 > .
  eq [ok] : isOk(L < X,Y >) = X >= 1 and Y >= 1 and X <= size and Y <= size
                              and not(< X,Y > in L) and not(< X,Y > in wall) .
  ceq [c1] : P in L = true if L1 P L2 := L .
  eq [c2] : P in L = false [owise] .

  rl [downN] : next(L < X,Y >, N) => < X,Y + N > .
  rl [leftN] : next(L < X,Y >, N) => < sd(X,N),Y > .
  rl [upN] : next(L < X,Y >, N) => < X,sd(Y,N) > .
  rl [rightN] : next(L < X,Y >, N) => < X + N,Y > .
  rl [eject] : {PY1, L1 < X,Y > , M1} || {PY2, L2 < X,Y >, M2} => empty .
  crl [exit] : {PY, L < X,X >, M} => {PY, exit, M} if X == size .
  crl [walk] : {PY, L, M} => {PY, L P, M + 1} if next(L,1) => P /\ isOk(L P) .
  crl [jump] : {PY, L, M} => {PY, L P, M + 2} if next(L,2) => P /\ isOk(L P) .
endm
```

Figure 1. Maude specification of the maze game

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[3] equational theory and R is a set of admissible conditional rules.

[3] Equational specifications in Maude can be theories in membership equational logic, which may include conditional membership axioms that are not addressed in this paper. Actually, membership axioms can interact with operator attributes such as `assoc` and `iter` in undesirable ways (Clavel et al., 2011), which can be a major difficulty for a tracing-based tool like ours to work correctly.

Example 2.1

Consider the Maude system module `MAZE` of Figure 1, which is inspired by the maze example in (Riesco et al., 2012). The module is delimited by the Maude keywords `mod` and `endm`, and it encodes a conditional rewrite theory that specifies a maze game in which multiple players (modeled as terms of sort `Player`) must reach a given exit point. Players may enter the maze at distinct entry points, and can move through the maze by walking or jumping. Furthermore, any collision between two players eliminates them from the game.

`MAZE` imports and makes use of the predefined Maude module `NAT`, which provides the equational definition for natural numbers together with some common built-in operators for their manipulation such as addition (`+`) and subtraction [4] (`sd`). The operators in the module signature are declared using the keyword `op`, while their types structure is specified using the keywords `sorts` and `subsorts`. Module variables are declared by means of the keyword `vars`. Roughly speaking, a maze is a `size` $\times$ `size` grid in which each maze position is specified by a pair of natural numbers `< X,Y >` of sort `Pos`.

The internal maze structure is defined through the equation `wall`, which explicitly defines those cells that represent the maze walls (see Figure 2). Each player's path [5] in the maze is described by a term of sort `List` that specifies a list of (pairwise distinct) positions by means of the usual constructor operator `nil` (empty list) and the associative, juxtaposition operator `__` whose unity element is `nil`.

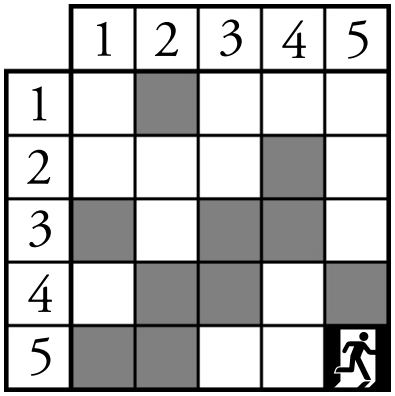

Figure 2: The 5×5 grid encoded in `MAZE` .

System states describe a game scenario by recording the paths taken by the different players from the moment they entered the game (at their respective entry points). We use the associative and commutative operator `||` (whose unity element is the constant `empty`) to model states as multisets of triples of the form $\{\mathtt{p}_1, \mathtt{L}_1, \mathtt{m}_1\}$ `||` ... `||` $\{\mathtt{p}_n, \mathtt{L}_n, \mathtt{m}_n\}$, $\mathtt{p}_i \neq \mathtt{p}_j$ for $\mathtt{i} \neq \mathtt{j}$, where $\mathtt{p}_i$ uniquely identifies the `i`th player, $\mathtt{L}_i$ is the path the `i`th player has hitherto followed since he entered the maze, and $\mathtt{m}_i$ is the length of the path $\mathtt{L}_i$ (which is different from the length of the list because players are allowed to jump two boxes in a single move), with $\mathtt{i} = 1, \ldots, \mathtt{n}$.

Given a player's path `L`, the next possible player's moves are nondeterministically computed by the rules `walk` and `jump`, which respectively augment `L` with the position `P` delivered by the rewrite expressions `next(L,N) => P`, with `N` = `1` (`walk`) or `N` = `2` (`jump`), occurring in the condition of these two rules. The function `next`(`L`, `N`) models all the possible `N`-cell movements that are available from the current player's location (given by the last position in `L`). In both rules, the correctness of the computed subsequent

[4] In order to avoid producing negative numbers, natural subtraction is implemented by using the symmetric difference operator (`sd`) that subtracts the smaller of its arguments from the larger.

[5] In our specification, only simple paths are considered (i.e., paths that do not contain loops), which amounts to saying that no position can be revisited by the same player twice.

position `P` is verified by means of the function `isOK(L P)`. Specifically, position `P` is valid iff it is within the limits of the maze, not repeated in `L`, and not a part of the maze wall. Note that the `jump` rule allows a player to leap over either a wall or another player provided the position reached is valid.

Collisions between two players are implemented by means of the `eject` rule, which checks whether two players bump on the same position and eliminates them from the maze by replacing their associated triples with the `empty` state value.

The `exit` rule checks whether a given player has reached the lower right corner position `< size,size >` that we assume to be the maze exit.

Finally, note that the `exit` and `eject` operations should be modeled by using equations rather than rewrite rules in order to provide appropriate, deterministic `exit` and `eject` behavior. Nonetheless, we deliberately specified them by using rules in order to illustrate the debugging capabilities of our forward exploration technique in Section 5.

The conditional slicing technique formalized in this article is formulated by considering the precise way in which Maude proves the conditional rewriting steps modulo an equational theory $E = \Delta \cup B$, which we describe in the following section (see Section 5.2 in (Clavel et al., 2011) for more details).

2.4. Rewriting in Conditional Rewrite 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 (Klop, 1992) to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ (Bruni and Meseguer, 2006). In other words, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is, in general, undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The Maude interpreter implements conditional rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$. These 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 $t\downarrow_{\Delta,B}$ to which no further equations can be applied. The term $t\downarrow_{\Delta,B}$ is called a *canonical form* of t w.r.t. Δ modulo B. 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 (Clavel et al., 2011).

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

Roughly speaking, a conditional rewrite step on the term t applies a rewrite rule/equation to t by replacing a *red*ucible (sub-)*ex*pression of t (namely $t_{|w}$), called the *redex*,

by its contracted version $\rho\sigma$, called the *contractum*, whenever the condition C is fulfilled. 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 $e\sigma{\downarrow_{\Delta,B}}{=_B}\ true$; a matching equation $p := t$ *evaluates to true* w.r.t. σ if $p\sigma =_B t\sigma{\downarrow_{\Delta,B}}$; 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 $u =_B p\sigma$.[6] 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 $s \rightarrow_{R/E} t$ modulo E on a term s can be implemented without loss of completeness by applying the following rewrite strategy (Durán and Meseguer, 2010):

(1) **Equational simplification of s in Δ modulo B**, that is, reduce s using $\rightarrow_{\Delta,B}$ until the canonical form w.r.t. Δ modulo B ($s\downarrow_{\Delta,B}$) is reached;
(2) **Rewrite** ($s\downarrow_{\Delta,B}$) **in R modulo B** to t' using $\rightarrow_{R,B}$, where $t' \in [t]_E$.

A *computation* (trace) $\mathcal{C}$ for s_0 in the conditional rewrite theory $(\Sigma, \Delta \cup B, R)$ is then deployed as the (possibly infinite) rewrite sequence

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

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. Note that, following this strategy, after each conditional rewriting step using $\rightarrow_{R,B}$, generally the resulting term s_i, $i = 1, \ldots, n$, is not in canonical normal form and is thus normalized before the subsequent rewrite step using $\rightarrow_{R,B}$ is performed. Also, in the precise strategy adopted by Maude, the last term of a finite computation is finally normalized before the result is delivered.

Therefore, any computation can be interpreted as a sequence of juxtaposed $\rightarrow_{R,B}$ and $\rightarrow^*_{\Delta,B}$ transitions, with an additional equational simplification $\rightarrow^*_{\Delta,B}$ (if needed) at the beginning of the computation as depicted below.

$$\overbrace{s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_{\Delta,B}} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} \underbrace{s_1{\downarrow_{\Delta,B}}}} \rightarrow_{R,B} s_2 \rightarrow^*_{\Delta,B} \underbrace{s_2{\downarrow_{\Delta,B}} \cdots}$$

We define a *Maude step* from a given term s as any of the sequences $s \rightarrow^*_{\Delta,B} s{\downarrow_{\Delta,B}} \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$ that head the non-deterministic Maude computations for s. Note that, for a canonical form s, a Maude step for s boils down to $s \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$. We define $m\mathcal{S}(s)$ as the set of all the non-deterministic Maude steps stemming from s.

3. Instrumented Computations

In this section, we introduce an auxiliary technique for instrumenting computations. The instrumentation allows the relevant information of the rewrite steps, such as the selected redex and the contractum produced by the step, to be traced explicitly despite

[6] 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 modulo B —i.e., a term p such that, for every substitution σ, if $x\sigma$ is a canonical form w.r.t. Δ modulo B for every $x \in Dom(\sigma)$, then $p\sigma$ is also a canonical form w.r.t. Δ modulo B.

the fact that terms are rewritten modulo a set B of equational axioms that may cause their components to be implicitly reordered. Given a computation $\mathcal{C}$, let us show how $\mathcal{C}$ can be expanded into an *instrumented* computation $\mathcal{T}$ in which each application of the matching modulo B algorithm that is used in $\rightarrow_{R,B}$-steps and $\rightarrow_{\Delta,B}$-steps is explicitly mimicked by the specific application of a bogus equational axiom, which is oriented from left to right and then applied as a rewrite rule in the standard way.

Typically hidden inside the B-matching algorithms, some pertinent term transformations allow terms that contain operators obeying equational axioms to be rewritten into supportive B-normal forms that facilitate the matching modulo B. In the case of AC-theories, these transformations allow terms to be reordered and correctly parenthesized in order to enable subsequent rewrite steps. Basically, this is achieved by producing a single, auxiliary representative of their AC congruence class (i.e., the AC-normal form). An AC-normal form is typically generated by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, sorting these arguments under some linear ordering and combining equal arguments using multiplicity superscripts (Eker, 2003). For example, the congruence class containing $f(f(\alpha, f(\beta, \alpha)), f(f(\gamma, \beta), \beta))$ where f is an AC symbol and subterms α, β, and γ belong to alien theories might be represented by $f^*(\alpha^2, \beta^3, \gamma)$, where f^* is a variadic symbol that replaces nested occurrences of f. A more formal account of this transformation is given in (Eker, 1995).

As for purely associative theories, we can get an A-normal form by just flattening nested function symbol occurrences without sorting the arguments. This case has practical importance because it corresponds to lists. C-normal forms are just obtained by properly ordering the arguments of a commutative binary operator. Finally, for function symbols that satisfy the unit axiom U, the unity element of U is not included in the U-normal form, and variables under a U symbol can always be assigned the unity element through U-matching (Eker, 1995).

Then, rewriting modulo B in Maude proceeds by using the special form of matching called B-matching on the internal representation of terms as B-normal forms, where B may contain, among others, any combination of associativity, commutativity, and unity axioms for different binary operators. Moreover, at each Maude step, the resulting term is shown in B-normal form (without multiplicity superscripts).

In the following, we discuss how we can simulate B-matching in our framework by means of specific "fake" axioms that mimick the B-matching transformation of terms that occur internally in Maude. This allows these transformations to be unhidden and explicitly revealed in the output trace. This artifice is only a means to reveal the term transformations of subterms that are forced by the step so that any position can be properly traced across rewriting steps.

Example 3.1

Consider a binary AC operator f together with a simple, standard lexicographic ordering over constant symbols. Given the term $f(b, f(f(b, a), c))$, let us reveal how this term matches modulo AC the left-hand side of the rule $[r] : f(f(x, y), f(z, x)) \Rightarrow x$ with AC-matching substitutions $\{x/b, y/a, z/c\}$ and $\{x/b, y/c, z/a\}$. For the first solution, this is mimicked by the transformation sequence $f(b, f(f(b, a), c)) \overset{\texttt{toACnf}}{\longrightarrow} f^*(a, b^2, c) \overset{\texttt{fromACnf}}{\longrightarrow} f(f(b, a), f(c, b))$, where 1) the first step corresponds to a term transformation that obtains the AC-normal form $f^*(a, b^2, c)$, and 2) the second step corresponds to the inverse, an unflattening transformation that delivers the AC-equivalent term $f(f(b, a), f(c, b))$

that syntactically matches the left-hand side of rule r with substitution $\{x/b, y/a, z/c\}$. Note that an alternative unflattening transformation is possible $f^*(a, b^2, c) \stackrel{\texttt{fromACnf}}{\longrightarrow} f(f(b, c), f(a, b))$, which actually delivers the second AC-matcher $\{x/b, y/c, z/a\}$. When several B-matchers exist, we only consider those that are effectively computed by means of the Maude internal rewriting strategy.

In our implementation, rewriting modulo B proceeds by using the standard form of B-matching on B-normal forms supported by Maude, where the left-hand sides of the rules are always normalized and the right-hand sides are (partially or totally) normalized when convenient (typically, when the unity element needs to be removed).

Example 3.2

Consider two binary AC operators f and g and the rules $[r_1] : f(c, b, a) \Rightarrow g(c, b, a)$ and $[r_2] : f(c, f(b, a)) \Rightarrow g(c, g(b, a))$, whose left-hand (resp. right-hand) sides are pairwise equivalent modulo B. When the specification that contains them is loaded, the two rules are respectively normalized by Maude into the B-equivalent rules $[r'_1] : f(a, b, c) \Rightarrow g(a, b, c)$ and $[r'_2] : f(a, b, c) \Rightarrow g(c, g(a, b))$. Note that the left-hand side $f(c, b, a)$ of r_1 is reordered as $f(a, b, c)$ in r'_1, whereas the left-hand side $f(c, f(b, a))$ of r_2 is not only reordered but also flattened as $f(a, b, c)$ in r'_2.

As for the right-hand sides of the rules, the right-hand side $g(c, b, a)$ of r_1 is reordered as $g(a, b, c)$ in r'_1 whereas the right-hand side of r_2 is not flattened in r'_2 and only the subterm at position 2 (i.e., $g(b, a)$) is reordered; hence, the whole term in the right-hand side of r_2 is neither ordered nor flattened in r'_2.

In the sequel, when no confusion can arise, we refer to a given program's rule and its corresponding, internally normalized version by using the same label.

Therefore, any given instrumented computation consists of a sequence of conditional rewrite steps using the conditional equations ($\rightarrow_\Delta$), conditional rewrite rules ($\rightarrow_R$), equational axioms, and (internal) B-matching transformations ($\rightarrow_B$). More precisely, each rewrite step $s \stackrel{r,\sigma,w}{\rightarrow}_{R,B} t$ (resp., $s \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,B} t$) is broken down into a rewrite sequence $s \rightarrow_B^* s' \stackrel{r,\sigma,w}{\rightarrow}_{R,\emptyset} t' \rightarrow_B^* t$ (resp., $s \rightarrow_B^* s' \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,\emptyset} t' \rightarrow_B^* t$), where $s' =_B s$ and s' syntactically matches the (normalized) left-hand side of the equation e or rule r that is applied in the considered rewrite step. We define the rewrite relation $\rightarrow_K$ as $\rightarrow_R \cup \rightarrow_\Delta \cup \rightarrow_B$. By *instrument*($\mathcal{C}$), we denote a function that takes a computation $\mathcal{C}$ and delivers its instrumented counterpart.

Example 3.3

Consider the rewrite theory of Example 2.1 together with the following computation that consists of a single Maude step, which makes a downward movement from the current position modeled by the one-element list `< 1,1 >`:

$$\mathcal{C} = \texttt{next(< 1,1 >,1)} \stackrel{\texttt{downN}}{\longrightarrow}_{R,B} \texttt{< 1,1+1 >} \stackrel{\texttt{builtIn(+)}}{\longrightarrow}_{\Delta,B} \texttt{< 1,2 >}$$

The corresponding instrumented computation $\mathcal{T}$, which is produced by *instrument*($\mathcal{C}$), is as follows:

$$\mathcal{T} = \texttt{next(< 1,1 >,1)} \stackrel{\texttt{fromAUnf}}{\longrightarrow}_{B} \texttt{next(nil < 1,1 >,1)} \stackrel{\texttt{downN}}{\longrightarrow}_{R} \texttt{< 1,1+1 >} \stackrel{\texttt{builtIn(+)}}{\longrightarrow}_{\Delta} \texttt{< 1,2 >}$$

where the first, extra step `next(< 1,1 >,1)` $\overset{\texttt{fromAUnf}}{\longrightarrow_B}$ `next(nil < 1,1 >,1)` has been added to describe the internal transformation that affixes the unity element `nil` of the AU list operator `__` to the one-element list `< 1,1 >`. This is simply formalized as a rewrite step by using the bogus axiom `rl [fromAUnf] : < 1,1 > => nil < 1,1 >`. This transformation enables the subsequent application of the rewrite rule `downN`.

Note that the instrumented version of the Maude step reveals that the rewrite rule `downN` is not actually applied into the initial term `next(< 1,1 >,1)`, but rather into the AU-equivalent term `next(nil < 1,1 >,1)`, which is chosen to syntactically match the left-hand side of the (already normalized) applied rule.

Nevertheless, in order to improve readability, we omit B-matching transformations and the evaluation of built-in operators [7] when displaying Maude steps (unless explicitly stated otherwise). This is consistent with the strategy adopted by Maude for the case of B-matching transformations, and it is the default option in our tool. As described in Section 6, by using the tool Anima, the user can choose to visualize either the simplified view of a rewrite step or the complete and detailed instrumented version of the step.

4. Exploring Computation Trees

Given a conditional rewrite theory $\mathcal{R}$, the transition space of all computations in $\mathcal{R}$ from the initial term s can be represented as a *computation tree*, [8] $\mathcal{T}_{\mathcal{R}}(s)$. RWL computation trees are typically large and complex objects to deal with because of the highly-concurrent, nondeterministic nature of rewrite theories. Also, their complete generation and inspection are generally not feasible since some of their branches may be infinite as they encode nonterminating computations.

Example 4.1

Consider the rewrite theory of Example 2.1 together with the system state

$$\{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \{\texttt{p2,< 3,5 >,1}\}$$

that specifies two initial configurations for two players `p1` and `p2` in the maze. In this case, the computation tree describes all of the possible trajectories that the two players `p1` and `p2` (respectively starting at positions `< 4,4 >` and `< 3,5 >`) can take when they move simultaneously in the same maze. The paths are built by repeated (and independent) applications of the `walk` and `jump` rules, while the `eject` and `exit` rules respectively implement the expulsion of colliding players and the output of game players who reach the exit. A fragment of the computation tree is shown in Figure 3. For simplicity, we have chosen to decorate tree edges only with the labels of the rules that have been applied at each rewrite step, while other information such as the computed substitution and the rewrite position are omitted in the depicted tree.

---

[7] Maude provides efficient (C-like) built-in operators such as the addition `(+)` of natural numbers. This is thanks to a built-in mechanism called `iter` (short for *iterated operator*), which permits the efficient manipulation of very large stacks of unary operators, and an efficient binary representation of unbounded natural number arithmetic (Clavel et al., 2011).

[8] In order to facilitate trace inspection, computations are visualized as trees, although they are internally represented by means of more efficient graph-like data structures that allow common subexpressions to be shared.

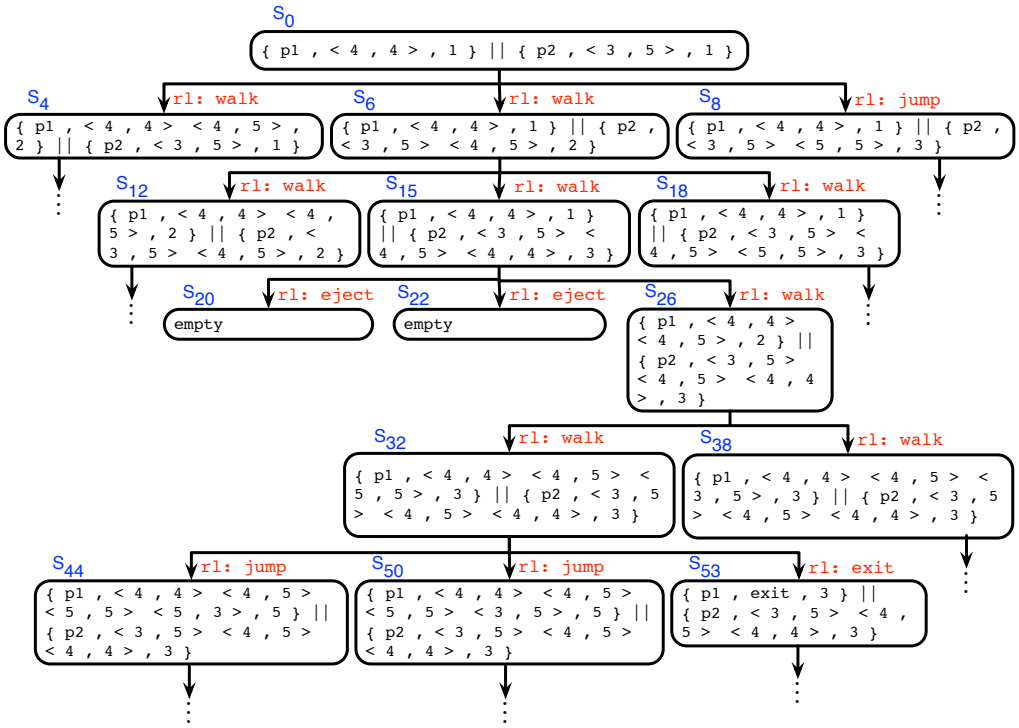

Figure 3. Computation tree

The instrumented version of a computation tree $\mathcal{T}_{\mathcal{R}}(s)$ can be constructed from $\mathcal{T}_{\mathcal{R}}(s)$ by expanding each computation in $\mathcal{T}_{\mathcal{R}}(s)$ into its corresponding instrumented counterpart as explained in Section 3. Also, it is possible to switch from the instrumented computation tree to the non-instrumented one by simply hiding the intermediate B-matching transformations and built-in evaluations that occur in the instrumented tree. In the sequel, we let $\mathcal{T}^{+}_{\mathcal{R}}(s)$ denote the instrumented computation tree that originates from the state s.

The rest of this section presents a generic, slicing-based exploration technique that allows the user to incrementally generate and inspect a fragment of the instrumented computation tree $\mathcal{T}^{+}_{\mathcal{R}}(s)$ by expanding (slices of) its computation states into their descendants starting from the root node. The exploration is an interactive procedure that can be completely controlled by the user, who is free to choose the computation states to be expanded. Roughly speaking, in our slices, certain subterms of a term are omitted, leaving "holes" that are denoted by special variable symbols.

4.1. Term Slices and Instrumented Computation Slices

A term *slice* of the term s is a term $s^{\bullet}$ that hides part of the information in s; that is, the irrelevant data in s that we are not interested in are simply replaced by special

$\bullet$-variables of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$. More precisely, in our framework, the term slice $s^\bullet$ can be seen as the term s plus a mask that hides the irrelevant symbols of s via $\bullet$-variables. In this way, the irrelevant part of $s^\bullet$ can be conveniently recovered from $s^\bullet$ at any time by simply unhiding the masked symbols, which we formalize using the function $\mathit{unhide}(s^\bullet) = s$ for every term slice $s^\bullet$ of s.

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

Basically, term slicing focuses on the information the user wants to observe of a given term. The next auxiliary definition formalizes the function $\mathit{Tslice}(t, P)$, which allows a term slice of t to be constructed w.r.t. a set of positions P of t. The function Tslice relies on the function $\mathit{fresh}^\bullet$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort that is distinct from any previously generated variable $\bullet_j$.

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

$$\mathit{Tslice}(t, P) = \mathit{recslice}(t, P, \Lambda), \text{ where}$$

$$\mathit{recslice}(t, P, p) = \begin{cases} f(\mathit{recslice}(t_1, P, p.1), \ldots, \mathit{recslice}(t_n, P, p.n)) & \\ \qquad \text{if } t{=}f(t_1, \ldots, t_n), n \geq 0, \text{ and } p \in \bar{P} & \\ t \qquad \text{if } t \in \mathcal{V} \text{ and } p \in \bar{P} & \\ \mathit{fresh}^\bullet \qquad \text{otherwise} & \end{cases}$$

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

Roughly speaking, the function $\mathit{Tslice}(t, P)$ yields a term slice of t w.r.t. a set of positions P that includes all symbols of t that occur within the paths from the root of t to any position in P, while each subterm $t_{|p}$, whose position p is minimal w.r.t. $\mathcal{P}os(t) \setminus \bar{P}$, is replaced by a freshly generated $\bullet$-variable.

Example 4.3

Consider the specification of Example 2.1 and initial state $t = \{\texttt{p1,< 4,4 >,1}\}$ `||` $\{\texttt{p2,< 3,5 >,1}\}$. Consider the set $P = \{1.1, 1.2.1, 1.2.2, 2.1\}$ of positions in t. Then,

$$\mathit{Tslice}(t, P) = \{\texttt{p1,< 4,4 >,} \bullet_1 \} \texttt{ || } \{\texttt{p2}, \bullet_2, \bullet_3\}$$

and the set of meaningful positions $\mathcal{MP}os(t^\bullet) = \{\Lambda, 1, 1.1, 1.2, 1.2.1, 1.2.2, 2, 2.1\}$.

Definition 4.4 (Inspection Criterion) *An* inspection criterion *is a function $\mathcal{I}(s^\bullet, s \to_K t)$ that, given a rewrite step $s \to_K t$ and a term slice $s^\bullet$ of s, computes a term slice $t^\bullet$ of t.*

Roughly speaking, inspection criteria allow us to control the information content conveyed by term slices resulting from the execution of $\to_K$-rewrite steps. It is worth noting that distinct implementations of the inspection criteria may produce distinct slices of the considered rewrite step. Several examples of inspection criteria are discussed in Section 5. We assume that the special value `fail` is returned by the inspection criterion whenever no slice $t^\bullet$ can be computed by $\mathcal{I}$.

$$\text{(frag)}\ \frac{V^\bullet = \mathcal{I}(U^\bullet, U \to V) \quad \wedge \quad V^\bullet \neq \mathsf{fail}}{\langle U \to V \to^* W, S^\bullet \bullet\!\!\to^* U^\bullet \rangle \Longrightarrow \langle V \to^* W, S^\bullet \bullet\!\!\to^* U^\bullet \bullet\!\!\to V^\bullet \rangle}$$

Figure 4. The inference rule frag of the transition system $(\mathit{Conf}, \Longrightarrow)$.

Given the instrumented computation $\mathcal{T} = (s_0 \to_K s_1 \cdots \to_K s_n)$, with $n \geq 1$, an *instrumented computation slice* of $\mathcal{T}$ w.r.t. the inspection criterion $\mathcal{I}$ is either the empty computation `nil` or the sequence $\mathcal{T}^\bullet_\mathcal{I} = (s^\bullet_0 \bullet\!\!\to s^\bullet_1 \bullet\!\!\to \cdots \bullet\!\!\to s^\bullet_n)$, which is generated by sequentially applying $\mathcal{I}$ to the steps that compose $\mathcal{T}$. We often write $\mathcal{T}^\bullet$ for an instrumented computation slice $\mathcal{T}^\bullet_\mathcal{I}$ when the inspection criterion $\mathcal{I}$ is clear from the context.

Let us formalize a calculus to generate instrumented computation slices by means of a transition system $(\mathit{Conf}, \Longrightarrow)$ (Plotkin, 2004) where:

- *Conf* is a set of *configurations* of the form $\langle \mathcal{T}, \mathcal{F}^\bullet \rangle$, where $\mathcal{T}$ is an instrumented computation and $\mathcal{F}^\bullet$ is an instrumented computation slice;
- the transition relation $\Longrightarrow$ implements the calculus of instrumented computation slices and is the smallest relation that satisfies the inference rule frag given in Figure 4.

Roughly speaking, the rule frag transforms the configuration $\langle U \to_K V \to^*_K W, S^\bullet \bullet\!\!\to^*_K U^\bullet \rangle$ into the configuration $\langle V \to^*_K W, S^\bullet \bullet\!\!\to^* U^\bullet \bullet\!\!\to V^\bullet \rangle$ where the first step $U \to_K V$ has been consumed and its corresponding slice $U^\bullet \bullet\!\!\to V^\bullet$ w.r.t. $\mathcal{I}$ has been added to $S^\bullet \bullet\!\!\to^* U^\bullet$. The rule frag only applies when the inspection criterion $\mathcal{I}$ generates a term slice $V^\bullet$ that is not the fail value.

The sequential application of the considered inference rule allows the instrumented computation $\mathcal{T}$ to be traversed in order to produce the sliced counterpart $\mathcal{T}^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}$. More formally,

Definition 4.5 (Instrumented Computation Slice) *Given the instrumented computation $\mathcal{T} = (s_0 \to_K s_1 \to_K \cdots \to_K s_n)$, with $n \geq 1$, the* instrumented computation slice $\mathcal{T}^\bullet$ *of $\mathcal{T}$ w.r.t. the inspection criterion $\mathcal{I}$ and term slice $s^\bullet_0$ of s_0 is defined by the function $\mathit{Cslice}(s^\bullet_0, \mathcal{T}, \mathcal{I})$, which is defined as follows.*

$$\mathit{Cslice}(s^\bullet_0, \mathcal{T}, \mathcal{I}) = \textbf{if } \langle \mathcal{T}, s^\bullet_0 \rangle \Longrightarrow^* \langle \texttt{nil}, \mathcal{T}^\bullet \rangle \textbf{ then } \mathcal{T}^\bullet \textbf{ else } \texttt{nil}$$

where `nil` *denotes the empty computation. Note that the second component $s^\bullet_0$ of the initial configuration $\langle \mathcal{T}, s^\bullet_0 \rangle$ matches the sequence $S^\bullet \bullet\!\!\to^* U^\bullet$ in rule frag by taking $s^\bullet_0$ for $U^\bullet$ and considering a sequence $S^\bullet \bullet\!\!\to^* U^\bullet$ consisting of zero steps.*

Computation slices can be *concretized* by replacing all the $\bullet$-variables that appear in the slices with suitable $\bullet$-free terms. More formally,

Definition 4.6 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and $\mathcal{T}$ be an instrumented computation in $\mathcal{R}$. Let $\mathcal{I}$ be an inspection criterion.*

Given the instrumented computation $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1 \overset{r_2,\sigma_2,w_2}{\to_K} \cdots \overset{r_n,\sigma_n,w_n}{\to_K} s_n)$, and the instrumented computation slice $\mathcal{T}^\bullet = (s^\bullet_0 \bullet\!\!\to s^\bullet_1 \bullet\!\!\to \cdots \bullet\!\!\to s^\bullet_n)$ of $\mathcal{T}$ w.r.t. $\mathcal{I}$ and a term slice $s^\bullet_0$, an instrumented computation slice concretization *$\mathcal{T}'$ of $\mathcal{T}^\bullet$ is any instrumented computation $s'_0 \overset{r_1,\sigma'_1,w_1}{\to_K} s'_1 \overset{r_2,\sigma'_2,w_2}{\to_K} \cdots \overset{r_n,\sigma'_n,w_n}{\to_K} s'_n$ such that each $s^\bullet_i$ is a term slice of $s'_i \in \tau(\Sigma, \mathcal{V})$, for all $i = 0, \ldots, n$.*

function *expand*$(s, s^\bullet, \mathcal{R}, \mathcal{I})$

1. $\mathcal{A} = \emptyset$
2. **for each** $\mathcal{M} \in m\mathcal{S}(s)$
3. $\quad \mathcal{M}^\bullet = \mathit{Cslice}(s^\bullet, \mathit{instrument}(\mathcal{M}), \mathcal{I})$
4. $\quad$ **if** $\mathcal{M}^\bullet \neq$ `nil` **then** $\mathcal{A} = \mathcal{A} \cup \{\mathcal{M}^\bullet\}$
5. **end**
6. **return** $\mathcal{A}$

endf

Figure 5. The one-step *expand* function.

4.2. Instrumented Computation Tree Slices

Instrumented computation tree slices are formally defined as follows.

Definition 4.7 (Instrumented Computation Tree Slice) *Let $\mathcal{T}^+_{\mathcal{R}}(s_0)$ be an instrumented computation tree for the term s_0 in the conditional rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$; let $s_0^\bullet$ be a term slice of s_0; and let $\mathcal{I}$ be an inspection criterion. An instrumented computation tree slice for $s_0^\bullet$ in $\mathcal{R}$ w.r.t. $\mathcal{I}$ is a tree $\mathcal{T}^+_{\mathcal{R},\mathcal{I}}(s_0^\bullet)$ (simply denoted by $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ when no confusion can arise) such that:*

(1) the root of $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ is $s_0^\bullet$;
(2) each branch of $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ is an instrumented computation slice $\mathcal{T}^\bullet$ w.r.t. $\mathcal{I}$ and $s_0^\bullet$ of a computation $\mathcal{T}$ in $\mathcal{T}^+_{\mathcal{R}}(s_0)$.
(3) for each instrumented computation $\mathcal{T}$ in $\mathcal{T}^+_{\mathcal{R}}(s_0)$, there is one, and only one, instrumented computation slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ in $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$.

In the following section, we show how tree slices of a given instrumented computation tree in $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ can be generated by repeatedly unfolding the nodes of the original tree.

4.3. Exploring the Computation Tree

In our methodology, instrumented computation tree slices are incrementally constructed by expanding tree nodes (i.e., term slices), starting from the root node (i.e., the initial term slice). Formally, given the term s and the term slice $s^\bullet$ of s, the expansion of s in the rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ w.r.t. the inspection criterion $\mathcal{I}$ is defined by the function *expand*$(s, s^\bullet, \mathcal{R}, \mathcal{I})$ of Figure 5, which unfolds the term slice $s^\bullet$ by deploying and then slicing all the possible instrumented Maude computation steps stemming from s that are given by $m\mathcal{S}(s)$. In other words, for each Maude step $\mathcal{M} = s \rightarrow^*_{\Delta,B} s{\downarrow_{\Delta,B}} \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$, we first compute its instrumented version and then the corresponding instrumented Maude step slice $\mathcal{M}^\bullet$ is generated, which is finally added to the set $\mathcal{A}$.

The overall construction methodology for instrumented computation tree slices is specified by the function *explore*, defined in Figure 6. Given a rewrite theory $\mathcal{R}$, a term slice $s_0^\bullet$ of the initial term s_0, and an inspection criterion $\mathcal{I}$, the function *explore* essentially

function *explore*$(s_0^\bullet, \mathcal{R}, \mathcal{I})$

1. $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet) = s_0^\bullet$
2. **while**$((s^\bullet = \mathit{pickLeaf}(\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet))) \neq \mathbf{EoE})$ **do**
3. $\quad \mathcal{T}^+_{\mathcal{R}}(s_0^\bullet) = \mathit{addPaths}(\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet), s^\bullet, \mathit{expand}(\mathit{unhide}(s^\bullet), s^\bullet, \mathcal{R}, \mathcal{I}))$
4. **od**
5. **return** $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$

endf

Figure 6. The interactive *explore* function.

formalizes an interactive procedure that is driven by the user starting from an elemental tree slice fragment, which only consists of the sliced root node $s_0^\bullet$. The instrumented computation tree slice $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ is built by choosing, at each loop iteration of the algorithm, the tree leaf that represents the term slice to be expanded by means of the auxiliary function $\mathit{pickLeaf}(\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet))$, which allows the user to freely select a leaf node from the frontier of the current tree $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$. Then, $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ is augmented by calling $\mathit{addPaths}(\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet), s^\bullet, \mathit{expand}(\mathit{unhide}(s^\bullet), s^\bullet, \mathcal{R}, \mathcal{I}))$, where $\mathit{unhide}(s^\bullet)$ recovers the original term s from $s^\bullet$. This function call adds all the instrumented computation slices w.r.t. $\mathcal{I}$ and $s^\bullet$ that correspond to the Maude steps that originate from the term s.

The special value **EoE** (End of Exploration) is used to terminate the inspection process: when the function $\mathit{pickLeaf}(\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet))$ is equal to **EoE**, no term to be expanded is selected and the exploration terminates delivering (a fragment of) the computation tree slice $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$.

5. Exploration Modalities

The methodology given in Section 4 provides a generic scheme for the exploration of (instrumented) computation trees w.r.t. a given inspection criterion $\mathcal{I}$ that must be selected or provided by the user. In this section, we show three implementations of the criterion $\mathcal{I}$ that produce three distinct exploration strategies. In the first case, the considered criterion allows an interactive program stepper to be derived in which conditional rewriting logic theories can be stepwisely animated. In the second case, we implement a partial stepper that allows computations with partial inputs to be stepped. Finally, in the last instantiation of the framework, the chosen inspection criterion implements an automated, forward slicing technique that simplifies the traces and allows relevant control and data information to be easily identified within the computation trees.

Since equations and axioms are both interpreted as rewrite rules in our formulation, throughout this section, notation $\lambda \Rightarrow \rho$ *if* C is often abused to denote rewrite rules as well as (oriented) equations and axioms.

5.1. Interactive Stepper

Given an instrumented computation tree $\mathcal{T}^+_{\mathcal{R}}(s_0)$ for an initial state s_0 and a conditional rewrite theory $\mathcal{R}$, the stepwise inspection of the computation tree can be directly

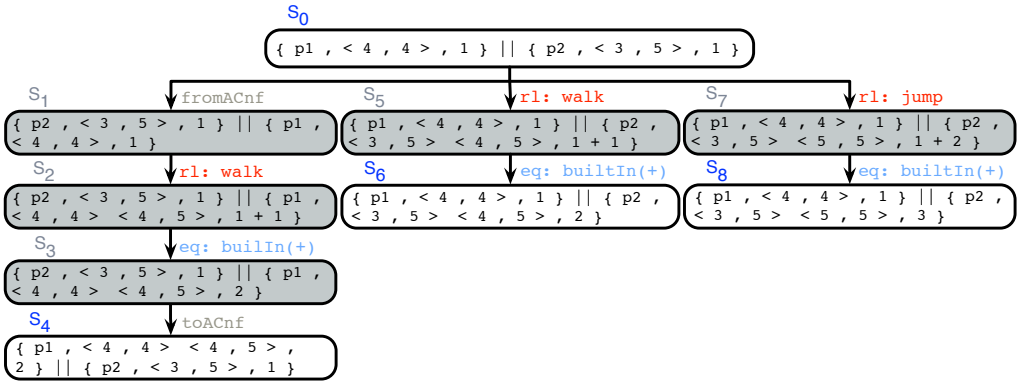

Figure 7. Inspection of the state s_0 w.r.t. $\mathcal{I}_{step}$.

implemented by instantiating the exploration scheme of Section 4 with the basic inspection criterion $\mathcal{I}_{step}(s, s \overset{r,\sigma,w}{\to}_K t) = t$, which simply returns the reduced term t of the rewrite step $s \overset{r,\sigma,w}{\to}_K t$. This way, by starting the exploration from a term slice that corresponds to the whole initial term s_0 (i.e., $s_0^\bullet = s_0$), the call $\mathit{explore}(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I}_{step})$ generates (a fragment of) the instrumented computation tree $\mathcal{T}^+_{\mathcal{R}}(s_0)$ whose topology depends on the program states that the user decides to expand during the exploration process.

Example 5.1

Consider the rewrite theory $\mathcal{R}$ in Example 2.1 and the computation tree in Example 4.1. Assume the user starts the exploration by calling $\mathit{explore}(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I}_{step})$, with $s_0 = s_0^\bullet$, which allows all the Maude steps that stem from the initial term s_0 to be expanded w.r.t. the inspection criterion $\mathcal{I}_{step}$. This generates the instrumented computation tree fragment $\mathcal{T}^+_{\mathcal{R}}(s_0)$ in Figure 7, where the instrumentation is made explicit. Now, the user can either quit or carry on with the exploration of nodes s_4, s_6, and s_8, which would result in the instrumented version of the tree fragment that is shown in Figure 3.

5.2. Partial Stepper

The computation states produced by the program stepper defined above do not include $\bullet$-variables. However, sometimes it may be useful to work with partial information and hence with term slices that "abstract some data" by using $\bullet$-variables. This may help the user focus on those parts of the program state that he/she wants to observe, while disregarding pointless information or unwanted rewrite steps.

We define the following inspection criterion

$$\mathcal{I}_{pstep}(s^\bullet, s \overset{r,\sigma,w}{\to}_K t) = \textbf{if } s^\bullet \overset{r,\sigma^\bullet,w}{\to}_K t^\bullet \textbf{ then } t^\bullet \textbf{ else } \mathsf{fail}$$

Roughly speaking, given a conditional rewrite step $s \overset{r,\sigma,w}{\to}_K t$, the criterion $\mathcal{I}_{pstep}$ returns a term slice $t^\bullet$ of the reduced term t, whenever $s^\bullet$ can be conditionally rewritten to $t^\bullet$ using the very same rule r at the same position w with the corresponding matching substitution $\sigma^\bullet$. The particularization of the exploration scheme given by the criterion $\mathcal{I}_{pstep}$ allows an interactive, partial stepper to be derived, in which the user can observe a distinguished

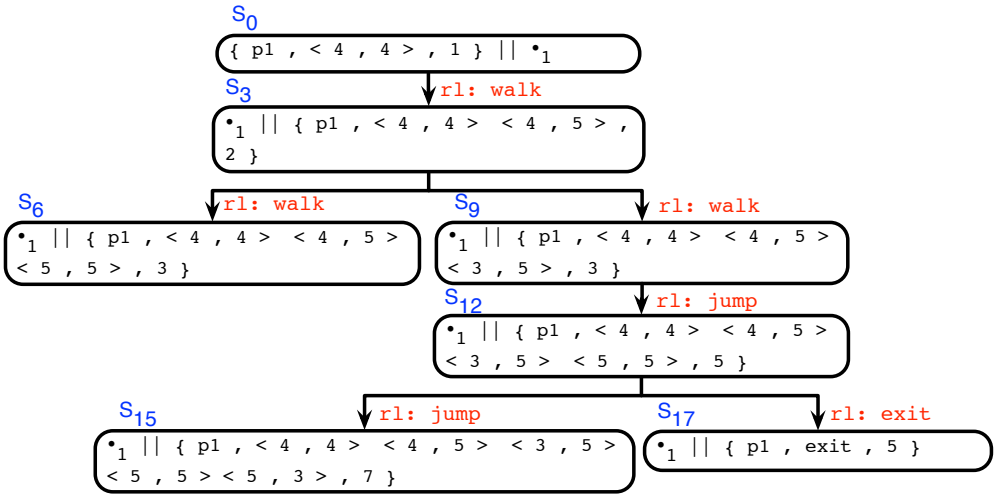

Figure 8. Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{pstep}$.

part of the state, thereby producing more compact and focused representations of the visited fragment of the (instrumented) computation tree.

The following example describes a simple partial stepper session. To improve its readability, here we omit the B-matching transformation steps and the calls to the `+` built-in operator.

Example 5.2

Consider the computation tree of Example 4.1 and the initial state

$$s_0 = \{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \{\texttt{p2,< 3,5 >,1}\}$$

Let $s_0^\bullet = \{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \bullet_1$ be a term slice of s_0 where only the triple structure of player `p1` is observed. Assume that the inspection criterion $\mathcal{I}_{pstep}$ is used to generate computation tree slice fragments. The computation tree slice fragment shown in Figure 8 is obtained by first expanding the node $s_0^\bullet$ into $s_3^\bullet$, then the node $s_3^\bullet$ into $s_6^\bullet$ and $s_9^\bullet$, then the node $s_9^\bullet$ into $s_{12}^\bullet$, and then the node $s_{12}^\bullet$ into $s_{15}^\bullet$ and $s_{17}^\bullet$. Note that the adopted partial stepping strategy allows a simplified view of (a part of) the considered computation tree to be constructed. More precisely, given the input encoded into the initial term slice $s_0^\bullet$, the computation can evolve by simply applying the rule `walk` to `p1` triple structure. By isolating `p1` movements in the tree slice fragment computed by partial stepping, the user can immediately observe and analyze why the player does not leave the game after reaching the exit but continues wandering for a while, which is an unnoticed side effect of declaring `exit` as a rule. To prevent `exit` from being nondeterministically applied in competition with other rules such as `walk` or `jump`, `exit` must be programmed as an equation to be deterministically used for normalizing the system state after the player has reached the maze exit. Note also that the `eject` rule does not appear in the tree slice fragment since player `p2` has been filtered out from the initial term slice $s_0^\bullet$ and therefore `eject` cannot be applied. Indeed, two players are required to match the left-hand side of

the considered rule.

5.3. Stepper and Partial Stepper Correctness

In this section, we provide a notion of correctness, which we call *universal correctness*, that holds for each instrumented computation slice computed by the stepper and the partial stepper described in Section 5.1 and Section 5.2, respectively. Universal correctness of an instrumented computation slice is formally defined as follows.

Definition 5.3 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\to_K} \cdots \overset{r_n,\sigma_n,w_n}{\to_K} s_n)$ be an instrumented computation in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}$ be an inspection criterion. Then, an instrumented computation slice $s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}$ and $s_0^\bullet$ is* universally correct *iff, for every instance s_0' of $s_0^\bullet$, there exists an instrumented computation slice concretization $s_0' \overset{r_1,\sigma_1',w_1}{\to_K} \cdots \overset{r_n,\sigma_n',w_n}{\to_K} s_n'$.*

Roughly speaking, Definition 5.3 ensures that, given an instrumented computation slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet)$, for every instance s_0' of $s_0^\bullet$, there exists an instrumented computation slice concretization $\mathcal{T}'$ of $\mathcal{T}^\bullet$ in which the rules involved in $\mathcal{T}^\bullet$ can be applied again, at the same positions. This amounts to saying that we can reproduce all the relevant information of $\mathcal{T}^\bullet$ in any rewrite sequence that instantiates $\mathcal{T}^\bullet$.

The following proposition states that non-empty instrumented computation slices that are generated using the inspection criterion $\mathcal{I}_{pstep}$ are always universally correct.

Proposition 5.4 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\to_K} \cdots \overset{r_n,\sigma_n,w_n}{\to_K} s_n)$ be an instrumented computation in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}_{pstep}$ be the partial stepper inspection criterion. Then, every instrumented computation slice $s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$ is universally correct.*

Proof. The proof proceeds by induction on the length n of $\mathcal{T}^\bullet$.

Base case: $n = 1$**.** Let us consider the one-step, instrumented computation slice $s_0^\bullet \bullet\!\!\to s_1^\bullet$. By Definition 4.5, $s_0^\bullet \bullet\!\!\to s_1^\bullet$ has been obtained by calling the function $\mathit{Cslice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1, \mathcal{I}_{pstep})$. Hence, $\mathcal{T}_1^\bullet = (s_0^\bullet \bullet\!\!\to s_1^\bullet) = \mathit{Cslice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1, \mathcal{I}_{pstep})$. This implies that the following transition $\langle s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1, s_0^\bullet \rangle \Longrightarrow^* \langle \mathtt{nil}, s_0^\bullet \bullet\!\!\to s_1^\bullet \rangle$ has been performed in the transition system $(\mathit{Conf}, \Longrightarrow)$ by means of one application of the frag rule. By definition of the frag rule (see Figure 4), $s_1^\bullet = \mathcal{I}_{pstep}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1)$. Now, by definition of $\mathcal{I}_{pstep}$, $s_1^\bullet = \mathcal{I}_{pstep}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1)$ iff $s_0^\bullet \overset{r_1,\sigma^\bullet,w_1}{\to_K} s_1^\bullet$. To complete the proof, it suffices to observe that the rewriting relation $\to_K$ is stable (i.e., it is closed under substitution applications). This amounts to saying that $s_0^\bullet\sigma' \overset{r_1,\sigma^\bullet\sigma',w_1}{\to_K} s_1^\bullet\sigma'$, for every substitution σ'.

Hence, for every instance $s_0' = s_0^\bullet\sigma'$ of $s_0^\bullet$, $s_0' \overset{r_1,\sigma^\bullet\sigma',w_1}{\to_K} s_1^\bullet\sigma' = s_1'$ is an instrumented computation slice concretization, which implies that $s_0^\bullet \bullet\!\!\to s_1^\bullet$ is a universally correct instrumented computation slice.

Inductive case: $n > 1$**.** Let us consider the instrumented computation slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\to s_1^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet)$, with $n > 1$, w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$. By the induction hypothesis, the instrumented computation slice $s_0^\bullet \bullet\!\!\to s_1^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_{n-1}^\bullet$ w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$ is universally

correct, that is, for every instance s'_0 of $s^\bullet_0$, there exists an instrumented computation slice concretization

$$s'_0 \overset{r_1,\sigma'_1,w_1}{\to_K} \cdots \overset{r_{n-1},\sigma'_n,w_{n-1}}{\to_K} s'_{n-1}. \tag{1}$$

Now, let us consider the last sliced step $s^\bullet_{n-1} \bullet\!\!\to s^\bullet_n$ of $\mathcal{T}^\bullet$. By applying an argument similar to the one in the base case, we can show that $s^\bullet_{n-1} \bullet\!\!\to s^\bullet_n$ is a universally correct instrumented computation slice. In other words, for every instance s''_{n-1} of $s^\bullet_{n-1}$, there exists an instrumented computation slice concretization

$$s''_{n-1} \overset{r_n,\sigma''_n,w_n}{\to_K} s''_n. \tag{2}$$

By choosing $s''_{n-1} = s'_{n-1}$, we can glue together the rewrite sequences 1 and 2 thereby obtaining that, for every instance s'_0 of $s^\bullet_0$, there exists an instrumented computation slice concretization

$$s'_0 \overset{r_1,\sigma'_1,w_1}{\to_K} \cdots \overset{r_{n-1},\sigma'_{n-1},w_{n-1}}{\to_K} s'_{n-1} \overset{r_n,\sigma'_n,w_n}{\to_K} s'_n. \tag{3}$$

Hence, $\mathcal{T}^\bullet$ is a universally correct instrumented computation slice w.r.t. $\mathcal{I}_{pstep}$ and $s^\bullet_0$. ■

Observe that a correctness result can be automatically obtained for the stepper inspection criterion $\mathcal{I}_{step}$ for free since $\mathcal{I}_{pstep}$ is a conservative generalization of $\mathcal{I}_{step}$, which allows $\bullet$-variables to appear in instrumented computation slices. In other words, $\mathcal{I}_{pstep}$ boils down to $\mathcal{I}_{step}$ when computation slices do not contain $\bullet$-variables. Therefore, since the stepper only works with $\bullet$-free instrumented computation slices (i.e., instrumented computations), the following result holds.

Corollary 5.5 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\to_K} \cdots \overset{r_n,\sigma_n,w_n}{\to_K} s_n)$ be an instrumented computation in the rewrite theory $\mathcal{R}$, with $n > 0$. Let $\mathcal{I}_{step}$ be the stepper inspection criterion. Then, $\mathcal{T}$ coincides with the instrumented computation slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{step}$ and $s^\bullet_0 = s_0$.*

Proof. Immediate by the fact that s_0 is a term that does not contain $\bullet$-variables, so $\mathcal{I}_{step}$ behaves as $\mathcal{I}_{pstep}$ and generates a universally correct instrumented computation slice $s_0 \bullet\!\!\to \cdots \bullet\!\!\to s_n$ by Proposition 5.4. The slice is unique as $\mathcal{I}_{step}$ cannot introduce $\bullet$-variables in the computation slice that can be bound to arbitrary terms. Therefore, $\mathcal{T}^\bullet$ is the very same $\mathcal{T}$. ■

For $\mathcal{I} \in \{\mathcal{I}_{step}, \mathcal{I}_{pstep}\}$, the universal correctness of instrumented computation slices can be easily lifted to universal correctness of instrumented computation tree slices as follows.

Theorem 5.6 (Universal Correctness) *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{I} \in \{\mathcal{I}_{step}, \mathcal{I}_{pstep}\}$. Let $s^\bullet_0$ be a term slice of term s_0. Let $\mathcal{T}^+_{\mathcal{R}}(s^\bullet_0)$ be (a fragment of) the instrumented computation tree slice in $\mathcal{R}$ w.r.t. $\mathcal{I}$ and $s^\bullet_0$ computed by the function explore$(s_0, s^\bullet_0, \mathcal{R}, \mathcal{I})$. Then, each branch $s^\bullet_0 \bullet\!\!\to \cdots \bullet\!\!\to s^\bullet_n$ in $\mathcal{T}^+_{\mathcal{R}}(s^\bullet_0)$ is a universally correct instrumented computation slice w.r.t. $\mathcal{I}$ and $s^\bullet_0$.*

Proof. Immediate. It suffices to apply Proposition 5.4 to each Maude step of each branch of $\mathcal{T}^+_{\mathcal{R}}(s^\bullet_0)$ when $\mathcal{I} = \mathcal{I}_{pstep}$ and to apply Corollary 5.5 when $\mathcal{I} = \mathcal{I}_{step}$. ■

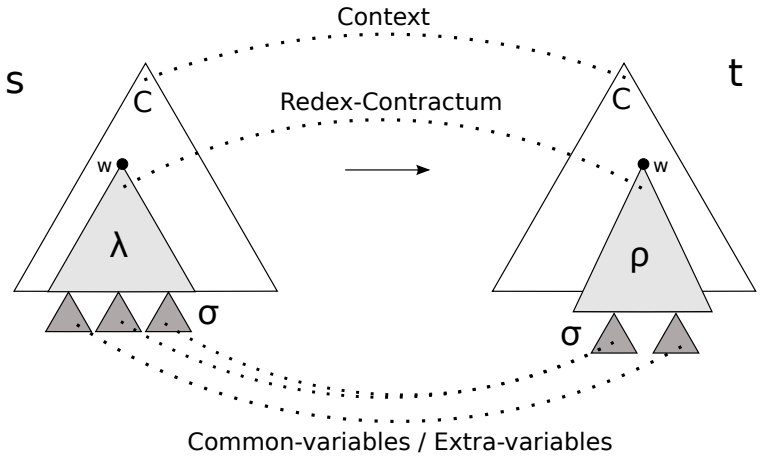

Figure 9. Meaningful descendants of a rewrite step $s \overset{r,\sigma,w}{\to}_K t$.

Note that universal correctness of an instrumented computation tree slice $\mathcal{T}^+_{\mathcal{R}}(s^\bullet_0)$ directly implies the universal correctness of the associated computation tree slice $\mathcal{T}_{\mathcal{R}}(s^\bullet_0)$. This is because $\mathcal{T}_{\mathcal{R}}(s^\bullet_0)$ is obtained from $\mathcal{T}^+_{\mathcal{R}}(s^\bullet_0)$ by simply removing those sliced steps that correspond to instrumentation transformations.

5.4. Forward Trace Slicer

Forward trace slicing is a program analysis technique that allows computations to be simplified w.r.t. a selected slice of their initial term. More precisely, given an instrumented computation $\mathcal{T}$ with initial term s_0 and a term slice $s^\bullet_0$ of s_0, forward slicing yields a simplified view $\mathcal{T}^\bullet$ of $\mathcal{T}$ in which each term s of the original instrumented computation is replaced by the corresponding term slice $s^\bullet$ that only records the information that depends on the meaningful symbols of $s^\bullet_0$, while irrelevant data are simply pruned away.

In the following, we define an inspection criterion $\mathcal{I}_{slice}$ that implements the forward trace slicing for a single conditional rewrite step. Given a conditional rewrite step $\mu = (s \overset{r,\sigma,w}{\to}_K t)$ and a term slice $s^\bullet$ of the term s, it delivers the term slice $t^\bullet$ that results from a dependency analysis of the meaningful information in $s^\bullet$ and the term transformation modeled by the rewrite rule r. During this analysis, the condition of the applied rule is recursively processed in order to ascertain the meaningful information that may depend on the conditional part of r.

Intuitively, our trace slicing algorithm enforces a notion of *meaningful* descendant that is as follows. Given a conditional rewrite step $\mu = (s \overset{r,\sigma,w}{\to}_K t)$ and a term slice $s^\bullet$ of s, the computed term slice $t^\bullet$ that contains the meaningful descendants of $s^\bullet$ by $\mathcal{I}_{slice}$ w.r.t. the rewrite step μ and $s^\bullet$ is given by the following relations, that are illustrated in Figure 9.

(1) (Context) Each meaningful symbol *sym* in $t^\bullet$ that occurs at a position w' such that $w' < w$, or w and w' are not comparable is a meaningful descendant of the very same symbol *sym* in $s^\bullet$.

(2) (Redex-contractum) Each meaningful symbol in $t^\bullet_{|w}$ that is introduced by the non-variable part of the right-hand side of the rule r is a meaningful descendant of all the meaningful symbols in $s^\bullet_{|w}$ that match the non-variable part of the left-hand side of r;

(3) (Common-variables) Each meaningful symbol sym in $t^{\bullet}_{|w}$ that is introduced by a binding x/t' of σ, for any variable x in the left-hand side of the rule r, is a meaningful descendant of the very same symbol sym in $s^{\bullet}_{|w}$;

(4) (Extra-variables) Each meaningful symbol in $t^{\bullet}_{|w}$ that is introduced by a binding z/t' of σ, where z is an extra-variable of the rule r, is a meaningful descendant of all those symbols in $s^{\bullet}_{|w}$ from which t' descends (to compute this dependency, the recursive inspection of the condition of r is required);

The meaningful descendant notion is illustrated in the following example.

Example 5.7

Consider the conditional rewrite rule `crl [r] : f(X,Y) => h(Z,X) if Z := g(X,Y)` that contains an extra-variable `Z` in its right-hand side, together with the equational definition `eq [sum] : g(X,Y) = X + Y`. Let us consider the one-step trace

$$\mathcal{T} = \texttt{c(f(2,3),a)} \rightarrow \texttt{c(h(5,2),a)}$$

that uses the rule `r` and equation `sum`.

In order to slice $\mathcal{T}$ w.r.t. the term slice $\texttt{c(f(2,}\bullet_1\texttt{),}\bullet_2\texttt{)}$, we need to analyze the internal computation trace $\mathcal{T}_{int} = \texttt{g(2,3)} \rightarrow \texttt{2 + 3} \rightarrow \texttt{5}$ for the evaluation of the matching condition $\texttt{Z} := \texttt{g(X,Y)}$, which instantiates the extra-variable `Z` to the value `5`.

The analysis determines that the computed value `5` in $\mathcal{T}_{int}$ descends from the input values `2` and `3` given to the variable arguments `X` and `Y` of the functions `g` and `f`. Thus, we conclude that the value `5` is a descendant (by the extra-variables relation) of the observed value `2` of the initial term slice $\texttt{c(f(2, }\bullet_1\texttt{), }\bullet_2\texttt{)}$. This is the key to compute the trace slice $\mathcal{T}^{\bullet} = \texttt{c(f(2, }\bullet_1\texttt{), }\bullet_2\texttt{)} \rightarrow \texttt{c(h(5,2), }\bullet_2\texttt{)}$ with the inspection function given in Figure 10.

Also note that symbol h is a descendant of the symbol f (by the redex-contractum relation), and the value `2` in the term slice $\texttt{c(h(5,2), }\bullet_2\texttt{)}$ descends from the input value `2` in $\texttt{c(f(2, }\bullet_1\texttt{), }\bullet_2\texttt{)}$ (by the common-variables relation). Finally, the root symbol `c` in $\texttt{c(h(5,2), }\bullet_2\texttt{)}$ descends from the same symbol `c` in $\texttt{c(f(2, }\bullet_1\texttt{), }\bullet_2\texttt{)}$ (by the context relation).

A precise formalization of the inspection criterion $\mathcal{I}_{slice}$ is provided by the function given in Figure 10. By adopting the inspection criterion $\mathcal{I}_{slice}$, the exploration scheme of Section 4 automatically turns into an interactive, forward trace slicer that expands computation states using the slicing methodology encoded into the inspection criterion $\mathcal{I}_{slice}$. In other words, given an instrumented computation tree $\mathcal{T}^{+}_{\mathcal{R}}(s_0)$ and a user-defined term slice $s^{\bullet}_0$ of the initial term s_0, any instrumented computation slice $s^{\bullet}_0 \bullet\!\!\rightarrow s^{\bullet}_1 \cdots \bullet\!\!\rightarrow s^{\bullet}_n$ in the tree $\mathcal{T}^{+}_{\mathcal{R}}(s^{\bullet}_0)$, which is computed by the *explore* function, is the forward sliced counterpart of an instrumented computation $s_0 \rightarrow_K s_1 \cdots \rightarrow_K s_n$ (w.r.t. the term slice $s^{\bullet}_0$) in the instrumented computation tree $\mathcal{T}^{+}_{\mathcal{R}}(s_0)$.

Roughly speaking, the inspection criterion $\mathcal{I}_{slice}$ works as follows. When the rewrite step $\mu = (s \overset{r,\sigma,w}{\rightarrow}_K t)$ occurs at a position w that is not a meaningful position of $s^{\bullet}$ (in symbols, $w \notin \mathcal{MP}os(s^{\bullet})$), trivially μ does not contribute to producing the term slice $t^{\bullet}$. Actually, the rewriting position w might not even occur in $s^{\bullet}$, hence we consider the prefix w' of w that points to a $\bullet$-variable in $s^{\bullet}$, i.e., $s^{\bullet}_{|w'}$ is a $\bullet$-variable. This position exists and is unique. Now, since no new relevant information descends from the term

function $\mathcal{I}_{slice}(s^\bullet, s \overset{r,\sigma,w}{\to}_K t)$ /**Assumption* : $[r] : \lambda \Rightarrow \rho$ if C*/

1. **if** $w \in \mathcal{MP}os(s^\bullet)$ **then**
2. $\quad \psi_0 = mgu(\lambda, (s^\bullet_{|w}))_{\restriction \mathcal{V}ar(\lambda)}$
3. $\quad$ **for** $i = 1$ **to** n **do** /**Assumption* : $C = c_1 \wedge \ldots \wedge c_n$*/
4. $\quad\quad \psi_i = process\text{-}condition(c_i, \sigma, \psi_{i-1})$
5. $\quad$ **od**
6. $\quad \theta = \{x/fresh^\bullet \mid x \in \mathcal{V}ar(r)\}$
7. $\quad t^\bullet = s^\bullet[\rho\psi_n\theta]_w$
8. **else**
9. $\quad t^\bullet = s^\bullet[fresh^\bullet]_{w'}$ $\quad$ with $w' \leq w \wedge s^\bullet_{|w'} = \bullet_i$, for some i
10. **fi**
11. **return** $t^\bullet$

endf

Figure 10. Inspection criterion that models the forward slicing of a conditional rewrite step.

slice $s^\bullet$, $\mathcal{I}_{slice}$ returns a variant $s^\bullet[fresh^\bullet]_{w'}$ of $s^\bullet$ where the subterm of $s^\bullet$ at the position w' has been replaced by a new fresh $\bullet$-variable that completely abstracts the contractum computed by μ.

Example 5.8

Consider the Maude specification of Example 2.1 and the following rewrite step

$$\mu : \{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \{\texttt{p2,< 3,5 >,1}\} \overset{\texttt{walk}}{\to}_K \{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \{\texttt{p2,< 3,5 > < 4,5 >,1+1}\}.$$

Let $s^\bullet = \{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \bullet_1$ be a term slice of $\{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \{\texttt{p2,< 3,5 >,1}\}$. Since the rewrite step μ occurs at position $2 \notin \mathcal{MP}os(\{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \bullet_1)$, which is not a meaningful position of $s^\bullet$, the inspection criterion $\mathcal{I}_{slice}$ returns the variant of $s^\bullet$ $\{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \bullet_2$, where $\bullet_2$ is a fresh variable generated by the function $fresh^\bullet$.

On the other hand, when $w \in \mathcal{MP}os(s^\bullet)$, the computation of $t^\bullet$ requires a more in-depth analysis of the conditional rewrite step that is based on a recursive slicing process that involves the conditions of the applied rule. This process is necessary for all descendants of $s^\bullet_{|w}$ in $t^\bullet$ to be properly tracked while any other information is disregarded.

More specifically, given the rewrite step $\mu : s \overset{r,\sigma,w}{\to}_K t$, with $[r] : \lambda \Rightarrow \rho$ if C, and the term slice $s^\bullet$, we proceed in two phases; in both phases, the $\bullet$-symbols in $s^\bullet$ are handled as *existentially quantified variables*, in contrast to the partial stepper of Section 5.2, where $\bullet$-symbols were interpreted to be universally quantified. The first phase retrieves the relevant information contained in the term slice $s^\bullet_{|w}$ of the redex $s_{|w}$, while the second phase recognizes relevant symbols that result from evaluating the rule condition (remind

function *process-condition*$(c,\ \sigma,\ \psi)$

1. **case** c **of**
2. $(p := m) \vee (m \Rightarrow p)$ **:** /* matching conditions and rewriting expressions */
3. **if** $(m\sigma = p\sigma)$
4. $\delta = mgu(p, m\psi)_{\restriction \mathcal{V}ar(p)}$
5. **else**
6. $((m\sigma)^{\bullet}\bullet\!\!\rightarrow^{+} (p\sigma)^{\bullet}) = Cslice(Tslice(m\sigma,\ \mathcal{MP}os(m\psi)), m\sigma \rightarrow_K^{+} p\sigma, \mathcal{I}_{slice})$
7. $\delta = mgu(p, (p\sigma)^{\bullet})_{\restriction \mathcal{V}ar(p)}$
8. **fi**
9. **return** $\delta\psi$
10. e **:** /* equational conditions */
11. **return** ψ
12. **end case**

endf

Figure 11. The condition processing function.

that Maude admits extra-variables that appear in the condition of an equation or rule while they do not appear in the corresponding left-hand side).

Phase 1. This phase first computes the most general unifier ψ_0 between the sliced redex $s^{\bullet}_{|w}$ and the left-hand side λ of the applied rule $[r] : \lambda \Rightarrow \rho$ if C restricted to the variables in λ. This allows the meaningful information of the sliced redex $s^{\bullet}_{|w}$ to be caught while those data that do not appear at meaningful positions are disregarded and not carried on by the rewrite step.

Note that the use of unification within our forward slicing methodology somehow resembles the unification-based, parameter-passing mechanism that is used in narrowing (Fay, 1979; Slagle, 1974). However, in our approach, we restrict unification to λ's variables and the computed unifiers are not propagated, by the inspection mechanism, to the whole term slice that we compute, but are only used to determine the relevant part of the contractum. Let us see an example.

Example 5.9

Consider the rewrite rule

```
rl [downN] : next(L < X,Y >, N) => < X,Y + N > .
```

in Example 2.1 together with the following rewrite step $\mathcal{C} =$ `next(< 1,1 >,1)` $\overset{\texttt{downN}}{\longrightarrow}_K$ `< 1,1+1 >` and the term slice `next(`$\bullet_1$`,1)`. In Phase 1, we compute the substitution ψ_0 such that

$$\psi_0 = mgu(\texttt{next(L < X,Y >, N)}, \texttt{next}(\bullet_1, 1))_{\restriction \{\texttt{L,X,Y,N}\}} = \{\texttt{N/1}\}.$$

Note that ψ_0 catches the meaningful value `1` for the variable `N` of the left-hand side of the rule `downN`.

It is worth noting that the most general unifier computation is essential for this kind of analysis and cannot be replaced by a simpler matching mechanism. Indeed, it could happen that there exists no matcher between the left-hand side λ of the rule and the term slice $s^{\bullet}_{|w}$, whereas an mgu can be computed. Hence, matching cannot guarantee that meaningful values for λ's variables can be detected.

Example 5.10

Consider Example 5.9. Note that the term $\texttt{next}(\bullet_1, \texttt{1})$ does not match the left-hand side `next(L < X,Y >, N)`. Hence, if we replaced the mgu computation in Example 5.9 by $match_{\texttt{next(L < X,Y >, N)}}(\texttt{next}(\bullet_1, \texttt{1}))$, we could not get the binding `N/1` that encodes the meaningful value `1` for the variable `N`.

Phase 2. This phase detects relevant information that originates from the rule condition. 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 $c_i\sigma$, $i = 1, \ldots, n$ by using the auxiliary function *process-condition* given in Figure 11 that generates a substitution ψ_i, such that ψ_i is used to further refine the partially ascertained substitution ψ_{i-1} that has been computed by incrementally analyzing the (sub)conditions $c_1\sigma, \ldots, c_{i-1}\sigma$.

When the whole condition $C\sigma$ has been processed, we get the substitution ψ_n, which basically encodes all the relevant instantiations discovered by analyzing the conditional rewrite step μ w.r.t. $s^{\bullet}_{|w}$.

Now, we consider the substitution $\theta = \{x/\mathit{fresh}^{\bullet} \mid x \in \mathcal{V}ar(r)\}$ that binds all the variables of the rule r to irrelevant information. By composing ψ_n with θ, we get a substitution that associates each variable of r with either a relevant or irrelevant value. Specifically, the composition $\psi_n\theta$ includes all the relevant bindings encoded in ψ_n plus the bindings of the form $x/\bullet$ for every variable x that is not in the domain of ψ_n. The term slice $t^{\bullet}$ is now computed from $s^{\bullet}$ by replacing its subterm at position w with the instance $(\rho\psi_n\theta\ )$ of the right-hand side of the applied rule r. This way, all the relevant/irrelevant information detected is transferred into the resulting sliced term $t^{\bullet}$.

The *process-condition* function handles matching conditions, rewrite expressions, and equational conditions differently. Specifically, the substitution ψ_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 $\mu : s \stackrel{r,\sigma,w}{\rightarrow}_K t$, recall that c is evaluated as follows: first, $m\sigma$ is reduced to its canonical form $m\sigma{\downarrow_{\Delta,B}}$, and then the condition $m\sigma{\downarrow_{\Delta}} =_B p\sigma$ is checked. In our framework, this corresponds to producing an internal instrumented computation that transforms $m\sigma$ into $p\sigma$.

The analysis of the matching condition $p := m$ during the slicing process of μ is implemented in *process-condition* by distinguishing the following two cases.

Case i. If $p\sigma = m\sigma$, then there is no need to generate the canonical form of $m\sigma$, since $p\sigma$ and $m\sigma$ are the same term. Hence, we discover new (possibly) relevant bindings for variables in p by computing the mgu δ between p and $m\psi$ restricted

to p's variables. Then, the algorithm returns the substitution composition $\delta\psi$ that updates the input substitution ψ with the new bindings encoded in δ.

Unlike most unification-based procedures, note that the resulting substitution $\delta\psi$ is obtained by composing the input substitution ψ with the updating substitution δ in reverse order of their computation (ψ was obtained before δ). This is because bindings of the form $x/\bullet$, possibly existing in ψ, must be overridden with meaningful bindings x/t, with $t \neq \bullet$, eventually appearing in the updating substitution δ. This is achieved by composing $\{x/t\}\{x/\bullet\}$, whereas the meaningful binding $\{x/t\}$ would be lost if we composed $\{x/\bullet\}\{x/t\}$. Since the co-domains of ψ and δ contain no variables apart from $\bullet$-symbols, the reverse composition ordering is harmless for any other bindings different from $x/\bullet$.

Case ii. When $p\sigma \neq m\sigma$, the slicing of the (internal) instrumented computation $\mathcal{T}_{int} = m\sigma \rightarrow_K^+ p\sigma$ is required. The slicing process is done by recursively invoking the function $Cslice(Tslice(m\sigma,\ \mathcal{MP}os(m\psi)), m\sigma \rightarrow_K^+ p\sigma, \mathcal{I}_{slice})$ whose outcome is the computation slice $(m\sigma)^\bullet \bullet\!\!\rightarrow^+ (p\sigma)^\bullet$ from which new relevant bindings for p's variables can be derived. This is done by computing the mgu δ between p and $(p\sigma)^\bullet$ restricted to the variables in p; then, the substitution composition $\delta\psi$ that updates ψ with these new bindings is returned.

– **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.** Unlike the evaluation of matching conditions and rewrite expressions, the equational conditions do not instantiate variables during the application of a rewrite step. This means that no new relevant instantiations can be detected for variables that appear in equational conditions. Therefore, in this case, the substitution, which is returned by *process-condition*, is just the input substitution ψ.

Example 5.11

Consider the Maude specification of Example 2.1 and the following rewrite step

$$\mu : \{\texttt{p1,< 4,4 >,1}\}\ \texttt{||}\ \{\texttt{p2,< 3,5 >,1}\} \overset{\texttt{walk},\sigma_{\texttt{walk}},2}{\rightarrow_K}$$
$$\{\texttt{p1,< 4,4 >,1}\}\ \texttt{||}\ \{\texttt{p2,< 3,5 >< 4,5 >,1+1}\}.$$

Let $\bullet_1$ || $\{\bullet_2, \texttt{< 3,5 >}, \bullet_3\}$ be a term slice of $\{\texttt{p1,< 4,4 >,1}\}$ || $\{\texttt{p2,< 3,5 >,1}\}$. Since the rewrite step μ occurs at position 2, which is a meaningful position of $\bullet_1$ || $\{\bullet_2, \texttt{< 3,5 >}, \bullet_3\}$, the two-phase procedure described above is applied.

Phase 1. The substitution ψ_0 is computed as follows.

$$\psi_0 = mgu(\{\texttt{PY,L,M}\}, \{\bullet_2, \texttt{< 3,5 >}, \bullet_3\})_{\restriction\{\texttt{PY,L,M}\}} = \{\texttt{PY}/\bullet_2, \texttt{L}/\texttt{< 3,5 >}, \texttt{M}/\bullet_3\},$$

where $\{\texttt{PY}, \texttt{L}, \texttt{M}\}$ is the left-hand side of the `walk` rule of Example 2.1. Note that the variable `L` in ψ_0 is bound to meaningful information, while `PY` and `M` are not considered to be relevant.

Phase 2. We first analyze the rewrite expression `next(L,1) => P` by calling the function *process-condition*(`next(L, 1) => P`, $\sigma_{\texttt{walk}}$, ψ_0). In this specific case, we have to consider the internal computation

$$\mathcal{T}_{int} = \texttt{next(< 3,5 >,1)} \overset{\texttt{rightN}}{\rightarrow_K} \texttt{< 3+1,5 >} \overset{\texttt{builtIn(+)}}{\rightarrow_K} \texttt{< 4,5 >}$$

whose computation slice w.r.t. $\mathcal{I}_{slice}$ and $\texttt{next}(\texttt{L}, 1)\psi_0$ coincides with $\mathcal{T}_{int}$, since

$$\texttt{next(L,1)}\psi_0 = \texttt{next(< 3,5 >,1)}.$$

Hence, $\delta = mgu(\texttt{P,< 4, 5 >})_{\restriction\{\texttt{P}\}} = \{\texttt{P/< 4,5 >}\}$. Observe that, by computing δ, we discover that the value bound to the variable `P` is meaningful. The evaluation of the condition `next(L,1) => P` terminates by returning the composition

$$\psi_1 = \delta\psi_0 = \{\texttt{P/< 4,5 >}\}\{\texttt{PY/}\bullet_2\texttt{,L/< 3,5 >,M/}\bullet_3\}$$

that updates the information of ψ_0 with the discovered binding in δ.

Subsequently, the condition `isOk(LP)` is processed. Since it is an equational condition, the function call *process-condition*$(\texttt{isOk(LP)}, \sigma_{\texttt{walk}}, \psi_1)$ returns a substitution ψ_2 such that $\psi_2 = \psi_1$, which implies that no new relevant information has been detected.

Finally, the term slice of $\{\texttt{p1,< 4,4 >,1}\}$ `||` $\{\texttt{p2,< 3,5 > < 4,5 >,1+1}\}$ is computed by replacing the subterm at position 2 of $\bullet_1$ `||` $\{\bullet_2\texttt{,< 3,5 >},\bullet_3\}$ with the instance $\{\texttt{PY,L P,M+1}\}\psi_2\theta$ of the right-hand side of the `walk` rule, where

$$\begin{aligned}\psi_2\theta = \psi_1\theta = \delta\psi_0\theta &= \{\texttt{P/< 4,5 >}\}\{\texttt{PY/}\bullet_2\texttt{,L/< 3,5 >,M/}\bullet_3\}\{\texttt{PY/}\bullet_4\texttt{,L/}\bullet_5\texttt{,M/}\bullet_6\texttt{,P/}\bullet_7\}\\ &= \{\texttt{P/< 4,5 >,PY/}\bullet_2\texttt{,L/< 3,5 >,M/}\bullet_3\}\end{aligned}$$

In symbols,

$$\bullet_1 \texttt{ || } \{\bullet_2\texttt{,< 3,5 >,1}\}[\{\texttt{PY,L P,M + 1}\}\psi_2\theta]_2 = \bullet_1 \texttt{ || } \{\bullet_2\texttt{,< 3,5 > < 4,5 >,}\bullet_3\texttt{+1}\}.$$

Since built-in operators are not provided in Maude with an explicit rule-based specification, the $\mathcal{I}_{slice}$ algorithm could not be directly applied to rewrite steps that involve these operators, hence we handle them in a special way. Given a built-in operator `op` and an instrumented computation $\mathcal{T}$, the idea is to handle every reduction `a op b` $\rightarrow$ `c` that occurs in $\mathcal{T}$ as an ordinary rewrite step, which is done by adding the extra equation `a op b = c` to the equational theory E considered for trace slicing. This way, every application of `op` that occurs in $\mathcal{T}$ is mimicked by applying its corresponding built-in equation, as we illustrate in the following example.

Example 5.12

Consider the rewrite step $\mu : \texttt{1+1} \overset{\texttt{builtIn}(+),\sigma_+,\Lambda}{\rightarrow_K} \texttt{2}$, and let $\bullet_1$`+1` be a term slice of `1+1`. Now, we first consider the supplementary ground equation `[builtIn(+)] 1+1 = 2` that explicitly models the reduction of the term `1+1`, and only then we compute $\mathcal{I}_{slice}(\bullet_1\texttt{+1}, \mu)$ that in this case yields the reduct `2`, thus producing the sliced step $\bullet_1$`+1` $\bullet\!\!\rightarrow$ `2`.

Note that, even if the first argument of the term slice $\bullet_1$`+1` is not considered meaningful, in the sliced step the computed reduct `2` is relevant. This happens because $\mathcal{I}_{slice}(\bullet_1\texttt{+1}, \mu)$ is obtained by using the *ground* equation `1+1 = 2` that preserves the constant contractum (i.e., the term `2`) in the generated term slice.

The following example describes the interactive construction of a fragment of an instrumented computation tree slice based on the $\mathcal{I}_{slice}$ criterion. The example also demonstrates how forward trace slicing can be fruitfully employed to debug RWL specifications. For the sake of readability, in the resulting computation tree slice fragment we omit all instrumentation steps, as in Example 5.2.

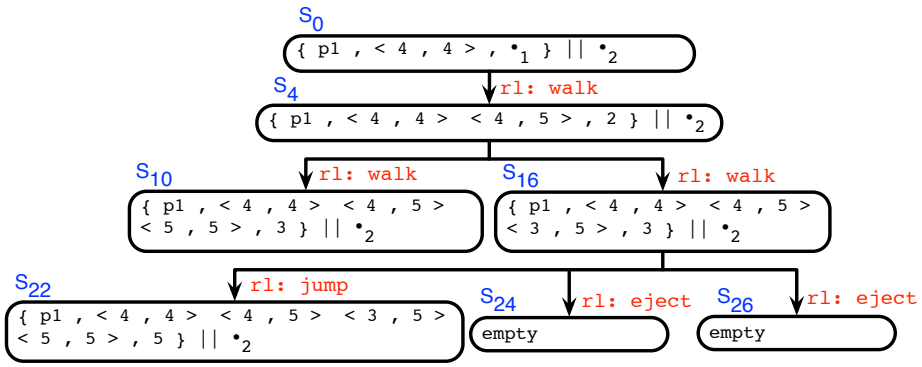

Figure 12. Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{slice}$.

Example 5.13

Consider the computation tree of Figure 3 whose initial term is $s_0 = \{$`p1,< 4,4 >,1`$\}$ `||` $\{$`p2,< 3,5 >,1`$\}$. Let $s_0^\bullet = \{$`p1,< 4,4 >,` $\bullet_1 \}$ `||` $\bullet_2$ be a term slice of s_0 where only player `p1` and its corresponding positions are observed. We get the computation tree slice fragment shown in Figure 12 by first expanding (w.r.t. the inspection criterion $\mathcal{I}_{slice}$) the node $s_0^\bullet$ into $s_4^\bullet$, then the node $s_4^\bullet$ into $s_{10}^\bullet$ and $s_{16}^\bullet$, and then the node $s_{16}^\bullet$ into $s_{22}^\bullet$, $s_{24}^\bullet$ and $s_{26}^\bullet$.

The slicing process automatically computes a computation tree slice fragment that represents a partial view of the maze game interactions from player `p1`'s perspective. Actually, irrelevant information is hidden and rules applied on irrelevant positions are directly ignored, which allows a simplified slice to be obtained thus favoring its inspection for debugging and analysis purposes. In fact, by isolating `p1` movements in the tree slice fragment computed by slicing, the user can immediately observe and debug the program. Specifically, by expanding the term slice $s_{16}^\bullet = \{$`p1,< 4,4 > < 4,5 > < 3,5 >,3`$\}$ `||` $\bullet_2$ into $s_{22}^\bullet$ by an application of the `jump` rule, and expanding $s_{16}^\bullet$ also into $s_{24}^\bullet$ and $s_{26}^\bullet$ by an application of the `eject` rule, the user can immediately realize that the player continues wandering for a while despite being ejected from the game, which clearly reveals the bug in the applied `eject` rule. To prevent `eject` from being nondeterministically applied in competition with other rules such as `walk` or `jump`, `eject` must be programmed as an equation to be deterministically used for normalizing the system state after a player is ejected. Note that the computation tree slice fragment shown in Figure 12 cannot be deployed by means of partial stepping since the chosen slicing criterion is overly restrictive to perform a partial rewrite step.

5.5. Forward Trace Slicer Correctness

Forward trace slicing produces computation slices that are generally not correct in the sense of Definition 5.3. This is because $\bullet_i$ symbols in the $\mathcal{I}_{slice}$ inspection criterion are interpreted as existential variables, while the partial stepper criterion $\mathcal{I}_{pstep}$ handles them as universally quantified variables. Let us see an example.

Example 5.14

Consider the rewrite theory that consists of the following rewrite rule $[r] : f(a, x) \Rightarrow x$, where a is a constant operator and x is a variable together with the rewrite step $\mu : f(a, b) \overset{r,\{x/b\},\Lambda}{\rightarrow_K} b$ and the term slice $f(\bullet_1, b)$ of $f(a, b)$. Then, we can compute the computation slice $\mu^\bullet$ of μ w.r.t. $\mathcal{I}_{slice}$ and $f(\bullet_1, b)$ by applying the function *Cslice*:

$$\mu^\bullet = \mathit{Cslice}(f(\bullet_1, b), \mu, \mathcal{I}_{step}) = f(\bullet_1, b) \bullet\!\!\rightarrow b.$$

Observe that $\mu^\bullet$ is not universally correct according to Definition 5.3. Indeed, for every instance of $f(\bullet_1, b)$ that replaces $\bullet_1$ with a term that is different from the constant a, there exists no computation slice concretization of $\mu^\bullet$.

Nonetheless, every computation $\mathcal{T}$ can be always reconstructed from a non-empty computation slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ by suitably instantiating all the $\bullet$-variables that appear in $\mathcal{T}^\bullet$. This allows us to formalize the following notion of existential correctness.

Definition 5.15 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} \cdots \overset{r_n,\sigma_n,w_n}{\rightarrow_K} s_n)$ be an instrumented computation in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}$ be an inspection criterion. Then, an instrumented computation slice $s_0^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}$ and $s_0^\bullet$ is* existentially correct iff every s_i is an instance of $s_i^\bullet$, with $i = 0, \ldots, n$.

Example 5.16

Consider the trace slice $\mu^\bullet = (f(\bullet_1, b) \bullet\!\!\rightarrow b)$ of $f(a, b) \overset{r,\{x/b\},\Lambda}{\rightarrow_K} b$ w.r.t. $\mathcal{I}_{slice}$ and $f(\bullet_1, b)$ in Example 5.14. Then, $\mu^\bullet$ is existentially correct.

The following proposition states that any non-empty computation slice, which is generated by means of the inspection criterion $\mathcal{I}_{slice}$, is existentially correct.

Proposition 5.17 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} \cdots \overset{r_n,\sigma_n,w_n}{\rightarrow_K} s_n)$ be an instrumented computation in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}_{slice}$ be the forward slicing inspection criterion. Then, every instrumented computation slice $s_0^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$ is existentially correct.*

Proof. Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} \cdots \overset{r_n,\sigma_n,w_n}{\rightarrow_K} s_n)$ be an instrumented computation in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}_{slice}$ be the forward slicing inspection criterion. Let us consider an arbitrary instrumented computation slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet)$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$. The proof proceeds by induction on the length n of $\mathcal{T}^\bullet$.

Base case: $n = 1$. Let us consider the one-step, instrumented computation slice $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet$ of $s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1$. We distinguish two cases.

Case $w_1 \in \mathcal{MP}os(s_0^\bullet)$. By hypothesis, $s_0^\bullet$ is a term slice of s_0, thus s_0 is an instance of $s_0^\bullet$ (in symbols, $s_0^\bullet \leq s_0$). Now, observe that $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet$ has been obtained by calling the function $\mathit{Cslice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1, \mathcal{I}_{slice})$. Hence, $\mathcal{T}_1^\bullet = (s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet) = \mathit{Cslice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1, \mathcal{I}_{slice})$. This implies that the following transition

$\langle s_0 \overset{r_1,\sigma_1,w_1}{\to}_K s_1, s_0^\bullet \rangle \Longrightarrow^* \langle \mathtt{nil}, s_0^\bullet \bullet\!\!\to s_1^\bullet \rangle$ has been performed in the transition system $(Conf, \Longrightarrow)$ by means of one application of the frag rule. By definition of the frag rule (see Figure 4), $s_1^\bullet = \mathcal{I}_{slice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\to}_K s_1)$. Now, by Definition of $\mathcal{I}_{slice}$, $s_1^\bullet = \mathcal{I}_{slice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\to}_K s_1) = s_0^\bullet[\rho\psi_m\theta]_{w_1}$, where ψ_m is the substitution obtained by applying the inspection criterion $\mathcal{I}_{slice}$ w.r.t. the rule $[r_1] : \lambda_1 \Rightarrow \rho_1$ if $c_1 \wedge \ldots \wedge c_m$, and $\theta = \{x/fresh^\bullet | x \in \mathcal{V}ar(r_1)\}$.

Now, it is immediate to prove (by a simple induction on m) that $\psi_m \leq \sigma_1$. Indeed, each binding in ψ_m either belongs to σ_1 or is of the form $x/\bullet_j$, for some natural number j. Moreover, $\psi_m\theta \leq \sigma_1$ as bindings in θ are all of the form $x/\bullet_j$. Hence,

$$\begin{aligned} s_1^\bullet = s_0^\bullet[\rho\psi_m\theta]_{w_1} &\leq s_0^\bullet[\rho\sigma_1]_{w_1} \quad (\text{by } \psi_m\theta \leq \sigma_1) \\ &\leq s_0[\rho\sigma_1]_{w_1} = s_1 \quad (\text{by } s_0^\bullet \leq s_0). \end{aligned}$$

This proves that $s_1^\bullet \leq s_1$. Finally, since $s_0^\bullet \leq s_0$ and $s_1^\bullet \leq s_1$, $s_0^\bullet \bullet\!\!\to s_1^\bullet$ is existentially correct.

Case $w_1 \notin \mathcal{MP}os(s_0^\bullet)$. In this case, $s_0^\bullet \bullet\!\!\to s_1^\bullet = s_0^\bullet[\bullet_f]_{w'}$ where $w' \leq w_1$, $s_{0|w'}^\bullet$ is a $\bullet$-variable, and $\bullet_f$ is a fresh $\bullet$-variable that has been generated by invoking $fresh^\bullet$. Again, by hypothesis, $s_0^\bullet \leq s_0$. Hence, there exists $\sigma_0^\bullet$ such that $s_0 = s_0^\bullet\sigma_0^\bullet$. Now, consider the substitution composition $\sigma_1^\bullet = \sigma_0^\bullet\{\bullet_f/s_{1|w'}\}$. Since $w' \leq w_1$, we have that $s_{1|w'}$ includes the contractum computed in the rewrite step $s_0 \overset{r_1,\sigma_1,w_1}{\to}_K s_1$. We immediately get $s_1^\bullet = s_0^\bullet[\bullet_f]_{w'} \leq s_0^\bullet\sigma_0^\bullet[\bullet_f\{\bullet_f/s_{1|w'}\}]_{w'} = s_0^\bullet[\bullet_f]_{w'}\sigma_1^\bullet = s_1$. Therefore, $s_i^\bullet \leq s_i$ for $i = 0, 1$, and we can conclude that $s_0^\bullet \bullet\!\!\to s_1^\bullet$ is existentially correct.

Inductive case: $n > 1$**.** Let us consider the instrumented computation slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\to s_1^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet)$, with $n > 1$, of $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\to}_K \cdots \overset{r_n,\sigma_n,w_n}{\to}_K s_n)$ w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$. By the induction hypothesis, the instrumented computation slice $s_0^\bullet \bullet\!\!\to s_1^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_{n-1}^\bullet$ w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$ is existentially correct; that is, for every instance $s_i^\bullet$, with $i = 0, \ldots, n-1$, $s_i^\bullet \leq s_i$. Now, let us consider the last sliced step $s_{n-1}^\bullet \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}^\bullet$. By proceeding similarly to the base case, we can show that $s_{n-1}^\bullet \bullet\!\!\to s_n^\bullet$ is an existentially correct instrumented computation slice of $s_{n-1} \overset{r_n,\sigma_n,w_n}{\to}_K s_n$ w.r.t. $\mathcal{I}_{slice}$ and $s_{n-1}^\bullet$. Therefore, we also have $s_n^\bullet \leq s_n$, which completes the proof.

■

The results in Proposition 5.17 can be directly lifted to (fragments of) instrumented computation tree slices that are generated by means of the forward slicing criterion $\mathcal{I}_{slice}$, thereby providing an existential correctness result for the overall forward slicing exploration technique.

Theorem 5.18 (existential correctness) *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{I}_{slice}$ be the forward slicing inspection criterion. Let $s_0^\bullet$ be a term slice of term s_0. Let $\mathcal{T}_{\mathcal{R}}^+(s_0^\bullet)$ be (a fragment of) the instrumented computation tree slice in $\mathcal{R}$ w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$ computed by the function explore$(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I}_{slice})$. Then, each branch in $\mathcal{T}_{\mathcal{R}}^+(s_0^\bullet)$ is an existentially correct instrumented computation slice w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$ of some instrumented computation in $\mathcal{R}$ that originates from s_0.*

Proof. Immediate. It suffices to apply Proposition 5.17 to each instrumented Maude step slice in each branch of $\mathcal{T}_{\mathcal{R}}^+(s_0^\bullet)$. ■

The existential correctness of the computation tree slice $\mathcal{T}_{\mathcal{R}}(s_0^\bullet)$ naturally derives from the existential correctness of its instrumented counterpart $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$. In fact, $\mathcal{T}_{\mathcal{R}}(s_0^\bullet)$ is obtained from $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ by hiding all the B-matching transformations and built-in evaluations that occur in the considered instrumentation.

6. Implementation

The exploration methodology developed in this paper has been implemented in the Anima tool, which is publicly available at `http://safe-tools.dsic.upv.es/anima/`. The underlying rewriting machinery of Anima is written in Maude and C++ and consists of about 250 Maude function definitions (approximately 2000 lines of source code) together with the implementation in C++ of `metaReducePath`, a new Maude command provided by our implementation that is described in Section 6.1. Anima also comes with an intuitive Web user interface based on AJAX technology, which allows users to graphically animate their programs and display fragments of computation trees. The core exploration engine is specified as a RESTful Web service by means of the Jersey JAX-RS API. The architecture of Anima is depicted in Figure 13 and consists of five main components: Anima Client, JAX-RS API, Anima Web Service, Database, and Anima Core. The Anima Client is purely implemented in HTML5 Canvas [9] and JavaScript. It represents the front-end layer of our tool and provides an intuitive, versatile Web user interface, which interacts with the Anima Web Service to invoke the capabilities of the Anima Core and save partial results in the MongoDB Database component, which is a scalable, high-performance, open source NoSQL database that perfectly fits our needs.

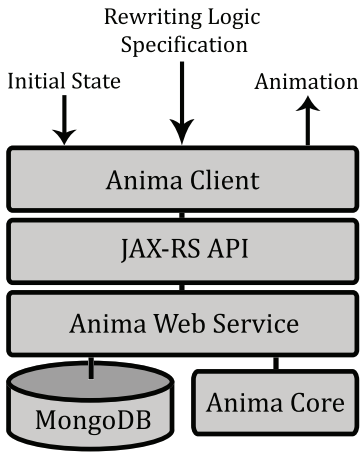

Figure 13: Anima architecture.

6.1. The metaReducePath command

One of the main challenges in the implementation of a trace-based Maude tool such as Anima is to make explicit the concrete sequence of internal term transformations occurring in a particular Maude computation, which is generally hidden and inaccessible within Maude's rewriting machinery. For the case of rule applications, this sequence can be easily retrieved by means of the Maude `metaSearchPath` command, but a similar command does not exist to ascertain the sequence of built-in operators and equations applied. These are only recorded in a raw text output trace, which cannot be manipulated as a meta-level expression by Maude. In order to overcome this drawback, we have

[9] For the sake of efficiency, browsers limit the maximum dimensions of a canvas object (e.g., Chrome limits a canvas to a maximum width or height of 8192 pixels). Exceeding these limits may cause the inability to properly display the current exploration.

implemented our own Maude command, named `metaReducePath`, which returns the detailed sequence of transformations (using equations, built-in operators, and any internal normalizations) applied to a term until its canonical form is reached.

The operator `metaReducePath` takes as arguments the metarepresentation $\bar{\mathcal{R}}$ of a system module $\mathcal{R}$ and the metarepresentation $\bar{t}$ of a term t. Its formal declaration is as follows.

```
sort ITrace ITraceStep .
subsort ITraceStep < ITrace .
op {_,_,_} : Equation Substitution Context -> ITraceStep [ctor] .
op metaReducePath : Module Term ~> ITrace? [special (...)] .
```

For a term t in $\mathcal{R}$, `metaReducePath(`$\bar{\mathcal{R}}$`,`$\bar{t}$`)` returns a term of sort `ITrace` that consists of a list of terms of sort `ITraceStep`, each of which is associated with a reduction step of the computation leading to the canonical form of t. The information recorded in `ITraceStep` terms can be accessed by means of the following observer functions:

```
op getEquation : ITraceStep -> Equation .
op getSubstitution : ITraceStep -> Substitution .
op getContext : ITraceStep -> Context .
```

More specifically, given a reduction step $s \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t$, these selectors respectively return: (i) the equation e, (ii) the substitution σ, and (iii) the *context*[10] surrounding the redex and in which the replacement takes place.

Maude is implemented thinking primarily of efficiency. However, this comes at the expense of subtle peculiarities of Maude's implementation that only became apparent during the development of Anima and that we addressed carefully. One of them shows up when the same equation is applied more than once in a single Maude step because a common redex appears more than once in the term, as illustrated in the following example that shows how Maude groups three applications of equation `EQ1` in a single "multi-reduction" step.

Example 6.1

Observe the reduction of the input term $\mathtt{g}(\mathtt{f}(\mathtt{a},\mathtt{b}),\mathtt{f}(\mathtt{a},\mathtt{b}),\mathtt{f}(\mathtt{a},\mathtt{b}))$ in the following functional module:

```
fmod EXAMPLE is
  sort Elem .
  ops a b c : -> Elem [ctor].
  op f : Elem Elem -> Elem .
  op g : Elem Elem -> Elem [ctor assoc comm] .
  vars X Y : Elem .
  eq [EQ1] : f(X,Y) = c .
endfm

Maude> reduce in EXAMPLE : g(f(a, b), f(a, b), f(a, b)) .
```

[10] A context is a term $C[\Box]$, with a hole $\Box$ at a distinguished position, that can be filled with a term.

```
********** equation
eq f(X, Y) = c [label EQ1] .
X --> a
Y --> b
f(a, b)
--->
c
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Elem: g(c, c, c)
```

These unorthodox *multi-steps* are the side effect of two efficient implementation optimizations, namely the alliance of identical subterms using multiplicity superscripts (as in $f(\alpha^2, \beta^3, \gamma)$) (Eker, 2003) and *in-place rewriting* (Eker, 2014), which means that equational rewriting destroys the DAG structure representing the term that is rewritten at each rewrite step. This is great for efficiency, but it is a major obstacle for exploring the computations. The new command `metaReducePath` provided by Anima detects and standardizes the *multi-steps* at the meta-level by spreading them out into the necessary number of single reduction steps, one for each single equation application, while explicitly recording the associated context information.

Example 6.2

For the specification and input term of Example 6.1, the standardized reduction trace that is obtained by invoking the command `metaReducePath` is as follows. [11]

```
Maude> reduce in META-LEVEL :
       metaReducePath(upModule('EXAMPLE, false),
       'g['f['a.Elem,'b.Elem],'f['a.Elem,'b.Elem],'f['a.Elem,'b.Elem]]) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result ITrace:
{eq 'f['X:Elem,'Y:Elem] = 'c.Elem [label('EQ1)] .,
  'X:Elem <- 'a.Elem ; 'Y:Elem <- 'b.Elem,
  'g[[],'f['a.Elem,'b.Elem],'f['a.Elem,'b.Elem]]}
{eq 'f['X:Elem,'Y:Elem] = 'c.Elem [label('EQ1)] .,
  'X:Elem <- 'a.Elem ; 'Y:Elem <- 'b.Elem,
  'g['c.Elem,[],'f['a.Elem,'b.Elem]]}
{eq 'f['X:Elem,'Y:Elem] = 'c.Elem [label('EQ1)] .,
  'X:Elem <- 'a.Elem ; 'Y:Elem <- 'b.Elem,
  'g['c.Elem,'c.Elem,[]]}
```

Technically, the execution of `metaReducePath` can be split into two phases: equational simplification and lifting to the meta-level. In the simplification phase, the input term is reduced to canonical form by using Maude's equational simplification. For each applied

[11] At the meta-level, constants are quoted identifiers that contain the constant's name and its type separated by a '.', (e.g., '0.Nat). Similarly, variables contain their name and type separated by a ':', (e.g., 'N:Nat). Composed terms are constructed in the usual way, by applying an operator symbol to a nonempty list of terms (Clavel et al., 2011).

equation and internal normalization transformation, our command additionally collects all the relevant information that we need to subsequently reconstruct the performed steps. This includes not only built-in evaluation but also memoization and other internal transformations such as the aforementioned `iter`, which replaces chains of iterations of a unary operator by a single instance of the iterated function, raised to the number of iterations, e.g., $\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$ as $\mathtt{s}^3(\mathtt{0})$. Once the term has reached its canonical form, the lifting phase consists of raising to the meta-level all the collected information and assembling the resulting instrumented computation.

	equational simplification		meta-level lifting		
n	rewrites	time (s.)	♮ size	$\lvert\mathcal{T}\rvert$	time (s.)
5	22	0	78	26	0
10	265	0	957	319	0
15	2,959	0.02	10,704	3,568	0.04
20	32,836	0.24	118,800	39,600	0.73
25	364,177	3.41	1,317,603	439,201	10.18

Table 1. Execution results of the `metaReducePath` command for `fibo(n)`.

Table 1 provides some figures regarding the execution of the `metaReducePath` command. We tested our command on a 3.3GHz Intel Xeon E5-1660 with 64GB of RAM by reducing different calls to the `fibo` function given in Figure 14. In Table 1, we distinguish the two phases mentioned above, namely equational simplification and lifting. For the equational simplification phase, the number of rewrites and the reduction times are given. For the lifting phase, we show the problem size, the length of the resulting instrumented computation, and the processing times. The problem size (column ♮ size) is measured as the number of expressions (applied equation, substitution, and context for each step) that are manipulated. The length of the resulting instrumented computation (column $\lvert\mathcal{T}\rvert$) is measured as the number of rewrite steps. Note that for extremely huge computations such as the trace of `fibo(25)`, which consists of 439,201 rewrite steps, the number of manipulated terms can be very high (more than 1,300,000) yet the execution time is reasonable (a few seconds) and comparable to existing Maude meta-commands that process millions of terms (Eker, 2003).

```
fmod FIBONACCI is pr NAT .
    op fibo : Nat -> Nat .
    var N : Nat .
    eq fibo(0) = 0 .
    eq fibo(1) = 1 .
    eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Figure 14: Benchmark problem for the `metaReducePath` command.

Finally, it is worth mentioning that `metaReducePath` takes into account the *Church-Rosser* and *termination* properties of functional modules assumed by Maude. Therefore, it returns just one possible simplification sequence that perfectly reproduces the normalization carried out by Maude following its internal strategy while ignoring the rest of the alternative normalizations.

6.2. Features

The main features of Anima include the following:

(1) *File uploading.* Maude specifications can be uploaded in Anima either as a simple *.maude* file or as a compressed *.zip* file, which must contain all the required files for the specification to work properly.

(2) *Inspection strategies.* The tool implements the three inspection strategies described in Section 5. As shown in Figure 15, the user can select the desired strategy by using the selector provided in the option pane.

(3) *Selection of meaningful symbols for slicing.* State slices can be specified by highlighting with the mouse the state symbols of interest directly on the nodes of the tree.

(4) *Expansion/Folding of program states.* The user can expand or fold states of the tree by left-clicking with the mouse on their state label, or by right-clicking with the mouse on the node and then selecting either the *Expand Node* option, the *Expand Subtree* option, or the *Fold Node* option that are offered in the contextual menu. The *Expand Subtree* option allows the user to automatically expand, up to a given depth k, for $k \leq 5$ (with default depth $k = 3$, which can be tuned by means of a slider), the subtree hanging from the considered node by following a breadth-first strategy.

When a state slice that is situated at the frontier of the computed tree slice fragment is selected for Expansion/Folding, the whole branch leading from the root of the tree to the selected node is highlighted, as illustrated in Figure 15. Common actions like dragging, zooming, and navigating the tree are allowed. Also, when a tree node is selected, the position of the tree on the screen is automatically rearranged to keep the chosen node at the center of the scene.

(5) *Display of instrumented steps.* The user can freely choose to display either a default, simplified view of a rewrite step (where only the applied rewrite rule is displayed), or the complete and detailed sequence of steps in the corresponding instrumented trace that simulates the step. This facility can be locally accessed by clicking on the $+/-$ symbols that respectively adorn the standard/instrumented view of the rewrite step, or by checking/unchecking the *Instrumented steps* option in the Anima option pane for the entire computation tree.

(6) *Tree Query mechanism.* The search facility illustrated in Figure 16 implements a pattern language that allows the selected information of interest to be searched in huge states of complex computation trees. The user only has to provide a filtering pattern (the query) that specifies the set of symbols that he/she wants to search for, and then all the states matching the query are automatically highlighted in the computation tree.

(7) *Showing rewrite step information.* Anima facilitates the inspection of any rewrite step $s \to t$ of the computation tree by underlining the differences between the two states (typically the selected redex of s and its contractum in t). In the case of a non-instrumented step $s \to_{\Delta,B} t$ (resp. $s \to_{R,B} t$), we generally cannot highlight the redex and contractum of the step as they might not exist in s and t because of the matching modulo B that precedes the rewrite step and the normalization that occurs after the rewrite step. Actually, recall that s and t are eventually reordered, augmented with unity elements, and parenthesised, yielding the B-equivalent terms

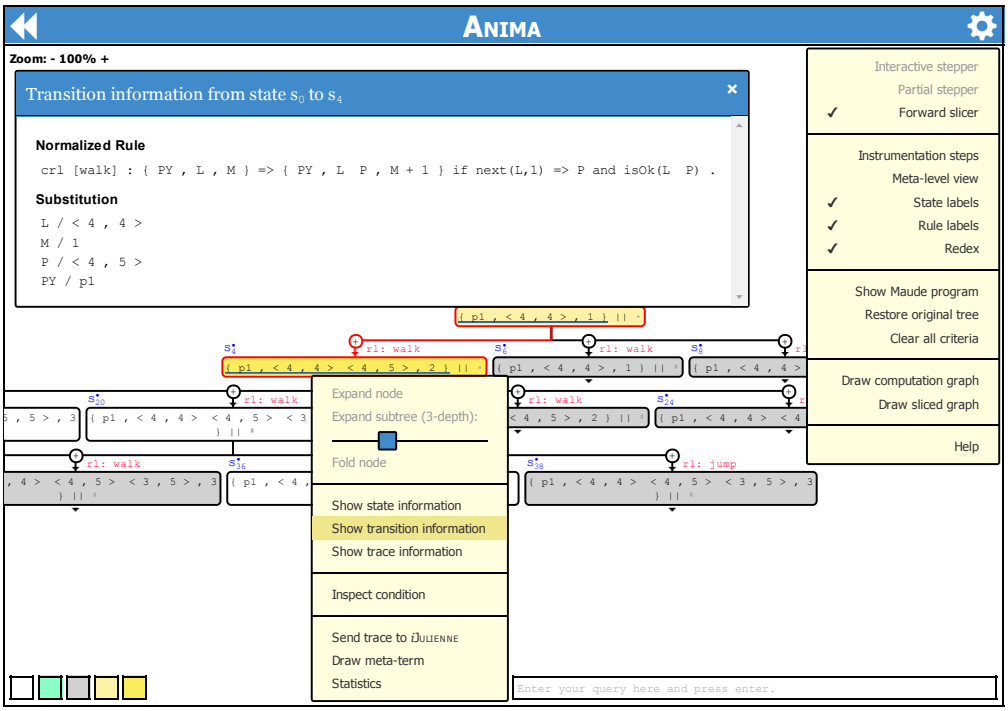

Figure 15. Anima at work.

s' and t' that star in an intermediate rewrite step $s' \to_\Delta t'$ (resp., $s' \to_R t'$). In this case, we underline the antecedents in s of the reduced redex in s' (and the descendants in t of the contractum that appears in t').

Furthermore, by clicking on the corresponding edge label of the tree, additional transition information is also displayed in the *transition information* window that shows up at the top, including the computed substitution and the normalized rule/equation applied.

(8) *Showing trace information.* By right-clicking a tree node and by selecting the *Show trace information* option, the user can obtain the complete information of the execution trace from the root to the selected node. This information is presented in a table that includes the labels of the rules and equations applied, the terms that result from the application of each rule or equation and the computed trace slice (if applicable) as shown in Figure 17. Moreover, Anima offers the possibility to export the displayed trace into meta-level representation, so the user can easily transfer the selected trace to any other Maude trace analyzer tool like the online backward trace analysis tool *i*JULIENNE (Alpuente et al., 2013c), for example.

(9) *Computation graph.* Even if the computation space for a given input term is hierarchically organized as a tree in order to systematize its exploration, Anima additionally supports the interactive inspection of a graph representation for the different space exploration modalities, namely (i) *computation graph*, which is available in all exploration modalities, (ii) *partial graph*, which is only available in the partial

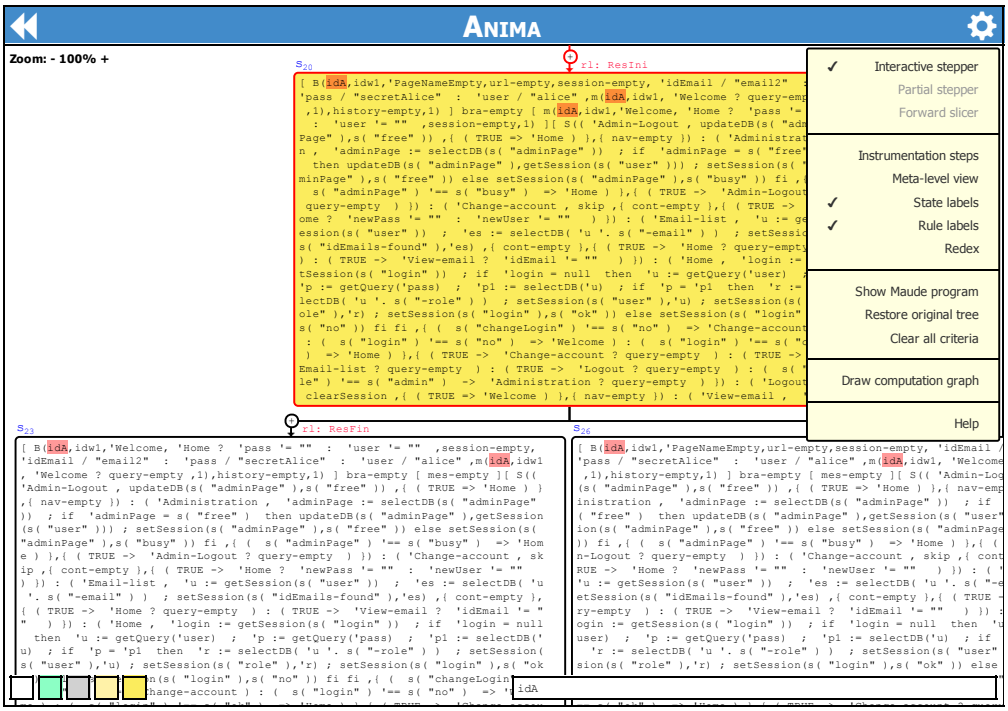

Figure 16. Anima search mechanism.

stepping modality, and (iii) *sliced graph*, which is only available in the forward slicing modality.

(10) *Graphic representation of meta-terms.* Anima facilitates the exhaustive inspection of any state of the computation tree by graphically representing the syntactic tree structure of its corresponding meta-term, including the exact position of each of its subterms.

(11) *Forward-Backward slicing integration.* In order to facilitate the exhaustive and incremental inspection of a given trace, Anima offers the possibility to export the trace to *i*Julienne (Alpuente et al., 2013c), which allows the origins or antecedents of a given expression (that is, those symbols in the initial state from which the observed expression descends) to be identified. This is done by tracing back all control and data dependencies.

Backward trace slicing can be achieved by right-clicking on a given state of the trace and then selecting the *Send trace to* *i*Julienne option. Reciprocally, *i*Julienne permits to export any state of the trace being inspected back to Anima, which accomplishes the full integration of both tools and greatly improves the trace inspection capabilities of our inspection frame.

(12) *Inspection of conditions.* As shown in Figure 18, Anima facilitates the inspection of the conditions satisfied during the application of a conditional rule or equation by right-clicking on the generated state and then selecting the *Inspect condition* option, which allows the user to export the traces deployed by evaluating the rule conditions to *i*Julienne for further analysis.

Trace information

Step	RuleName	Execution trace
1	'Start	{ p1 , < 4 , 4 > , 1 } || { p2 , < 3 , 5 > , 1 }
2	walk	{ p1 , < 4 , 4 > , 1 } || { p2 , < 3 , 5 > < 4 , 5 > , 1 + 1 }
3	builtIn	{ p1 , < 4 , 4 > , 1 } || { p2 , < 3 , 5 > < 4 , 5 > , 2 }
4	walk	{ p1 , < 4 , 4 > , 1 } || { p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 2 + 1 }
5	flattening	{ p1 , < 4 , 4 > , 1 } || { p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 1 + 2 }
6	builtIn	{ p1 , < 4 , 4 > , 1 } || { p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 3 }
7	unflattening	{ p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 3 } || { p1 , < 4 , 4 > , 1 }
8	walk	{ p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 3 } || { p1 , < 4 , 4 > < 4 , 5 > , 1 + 1 }
9	builtIn	{ p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 3 } || { p1 , < 4 , 4 > < 4 , 5 > , 2 }
10	flattening	{ p1 , < 4 , 4 > < 4 , 5 > , 2 } || { p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 3 }
11	unflattening	{ p1 , < 4 , 4 > < 4 , 5 > , 2 } || { p2 , < 3 , 5 > < 4 , 5 > < 5 , 5 > , 3 }
12	exit	{ p1 , < 4 , 4 > < 4 , 5 > , 2 } || { p2 , exit , 3 }
Total size:		840

Figure 17. Anima trace information.

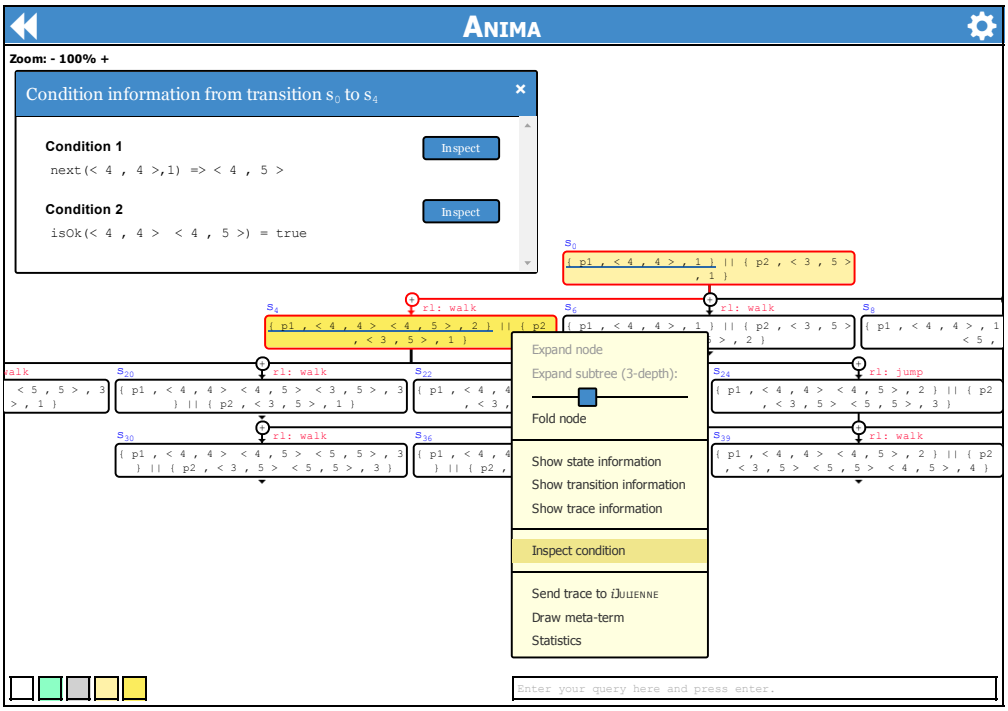

Figure 18. Inspection of a condition with Anima.

(13) *Showing statistics.* Finally, detailed statistics of the current computation tree can be accessed by selecting the *Statistics* option that appears in the contextual menu for any node in the tree. This shows, among others, the number of terms (normalized or not) that are reachable from this node, its number of children and depth in the tree, and the global size of the computation tree.

7. Conclusions

This paper presents a rich and highly dynamic, parameterized technique for the trace inspection of conditional rewrite theories that allows the nondeterministic execution of a given RWL theory to be followed up in different ways. Our technique is based on a generic animation algorithm that can be tuned to work with different modalities, including *incremental stepping* and *automated forward slicing*, which drastically reduces the size and complexity of the computations under examination. The algorithm is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates conditional RWL theories that involve rewriting modulo associativity (A), commutativity (C), and unity (U) axioms.

The proposed methodology is implemented and tested in the graphical tool Anima, which provides a skillful and highly dynamic interface for the dynamic analysis of RWL computations. This makes Anima 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 tool is useful for Maude programmers in two ways. First, it graphically exemplifies the semantics of the language, allowing the evaluation rules to be observed in action. Secondly, it can be used as a debugging tool, allowing the users to step forward and backward while slicing the trace in order to validate input data or locate programming mistakes. 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 applying a conditional equation, algebraic axiom, or rule, which greatly improves the standard view of execution traces in Maude and their meta-representations.

As already mentioned, the present version supports the instrumentation of matching modulo associativity, commutativity, and (left-, right- or two-sided) unity. In addition, Anima has an extensible design so that instrumentation for other equational axioms such as idempotency can be easily added in the future. We are also interested to extend our exploration technique to more sophisticated rewrite theories that may include membership axioms.

As for future work, we plan to exploit the dynamic dependencies exposed by our conditional trace slicing methodology to endow Anima with a program slicing capability that can identify those parts of a Maude theory that can (potentially) affect the values computed at some point of interest (Tip, 1995; Field and Tip, 1994).

References

Abadi, M., Lampson, B., Lévy, J.-J., 1996. Analysis and Caching of Dependencies. ACM SIGPLAN Notices 31 (6), 83–91.

Alpuente, M., Ballis, D., Baggi, M., Falaschi, M., 2010a. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In: Proceedings of the 19th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010). Association for Computing Machinery, pp. 43–52.

Alpuente, M., Ballis, D., Espert, J., Frechina, F., Romero, D., 2011. Debugging of Web Applications with WEB-TLR. In: Proceedings of the 7th International Workshop on Automated Specification and Verification of Web Systems (WWV 2011). Vol. 61 of Electronic Proceedings in Theoretical Computer Science (EPTCS). Open Publishing Association, pp. 66–80.

Alpuente, M., Ballis, D., Espert, J., Romero, D., 2010b. Model-checking Web Applications with Web-TLR. In: Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010). Vol. 6252 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 341–346.

Alpuente, M., Ballis, D., Falaschi, M., Frechina, F., Romero, D., 2013a. Rewriting-based Repairing Strategies for XML Repositories. The Journal of Logic and Algebraic Programming 82 (8), 326–352.

Alpuente, M., Ballis, D., Falaschi, M., Romero, D., 2006. A Semi-Automatic Methodology for Reparing Faulty Web Sites. In: Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006). IEEE Computer Society Press, pp. 31–40.

Alpuente, M., Ballis, D., Frechina, F., Romero, D., 2012a. Backward Trace Slicing for Conditional Rewrite Theories. In: Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2012). Vol. 7180 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 62–76.

Alpuente, M., Ballis, D., Frechina, F., Romero, D., 2012b. Julienne: A Trace Slicer for Conditional Rewrite Theories. In: Proceedings of the 18th International Symposium on Formal Methods (FM 2012). Vol. 7436 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 28–32.

Alpuente, M., Ballis, D., Frechina, F., Romero, D., 2014a. Using Conditional Trace Slicing for improving Maude Programs. Science of Computer Programming 80, Part B, 385 – 415.

Alpuente, M., Ballis, D., Frechina, F., Sapiña, J., 2013b. Parametric Exploration of Rewriting Logic Computations. In: Proceedings of the 5th International Symposium on Symbolic Computation in Software Science (SCSS 2013). Vol. 15 of EasyChair Proceedings in Computing (EPiC). EasyChair, pp. 4–18.

Alpuente, M., Ballis, D., Frechina, F., Sapiña, J., 2013c. Slicing-Based Trace Analysis of Rewriting Logic Specifications with *i*Julienne. In: Proceedings of the 22nd European Symposium on Programming (ESOP 2013). Vol. 7792 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 121–124.

Alpuente, M., Ballis, D., Frechina, F., Sapiña, J., 2014b. Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way). In: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014). Vol. 8373 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 229–255.

Alpuente, M., Ballis, D., Romero, D., 2009. Specification and Verification of Web Applications in Rewriting Logic. In: Proceedings of the 16th International Symposium on Formal Methods (FM 2009). Vol. 5850 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 790–805.

Alpuente, M., Ballis, D., Romero, D., 2014c. A Rewriting Logic Approach to the Formal Specification and Verification of Web Applications. Science of Computer Programming 81, 79–107.

Baader, F., Snyder, W., 2001. Unification Theory. In: Robinson, J. A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Vol. I. Elsevier Science, pp. 447–533.

Baggi, M., Ballis, D., Falaschi, M., 2009. Quantitative Pathway Logic for Computational Biology. In: Proceedings of the 7th International Conference on Computational Methods in Systems Biology (CMSB 2009). Vol. 5688 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 68–82.

Bethke, I., Klop, J. W., Vrijer, R. d., 2000. Descendants and Origins in Term Rewriting. Information and Computation 159 (1–2), 59–124.

Bruni, R., Meseguer, J., 2006. Semantic Foundations for Generalized Rewrite Theories. Theoretical Computer Science (TCS) 360 (1–3), 386–414.

Cheney, J., Ahmed, A., Acar, U. A., 2011. Provenance as Dependency Analysis. Mathematical Structures in Computer Science 21 (6), 1301–1337.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2007. All About Maude: A High-Performance Logical Framework. Springer-Verlag.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2011. Maude Manual (Version 2.6). Tech. rep., SRI International Computer Science Laboratory, available at: `http://maude.cs.uiuc.edu/maude2-manual/`.

Clements, J., Flatt, M., Felleisen, M., 2001. Modeling an Algebraic Stepper. In: Proceedings of the 10th European Symposium on Programming (ESOP 2001). Vol. 2028 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 320–334.

Durán, F., Meseguer, J., 2010. A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In: Proceedings of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA 2010). Vol. 6381 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 86–103.

Eker, S., 1995. Associative-Commutative Matching via Bipartite Graph Matching. The Computer Journal 38 (5), 381–399.

Eker, S., 2003. Associative-Commutative Rewriting on Large Terms. In: Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003). Vol. 2706 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 14–29.

Eker, S., 2014. Personal Communication.

Fay, M., 1979. First Order Unification in an Equational Theory. In: Proceedings of the 4th International Conference on Automated Deduction (CADE 1979). Academic Press, Inc., pp. 161–167.

Field, J., Tip, F., 1994. Dynamic Dependence in Term rewriting Systems and its Application to Program Slicing. In: Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP 1994). Vol. 844 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 415–431.

Huet, G., Lévy, J.-J., 1970. Call by need Computations in Nonambiguous Linear Term Rewriting Systems. Tech. Rep. 359, INRIA.

Klop, J., 1990. Term Rewriting Systems. Tech. Rep. CS-R9073, Centre for Mathematics and Computer Science.

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

Martí-Oliet, N., Meseguer, J., 2002. Rewriting Logic: Roadmap and Bibliography. Theoretical Computer Science (TCS) 285 (2), 121–154.

Meseguer, J., 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science (TCS) 96 (1), 73–155.

O'Donnell, M. J., 1977. Computing in Systems Described by Equations. Vol. 58 of Lecture Notes in Computer Science (LNCS). Springer-Verlag.

Plotkin, G. D., 2004. The Origins of Structural Operational Semantics. The Journal of Logic and Algebraic Programming 60–61 (1), 3–15.

Riesco, A., Asavoae, I. M., Asavoae, M., 2013. A generic program slicing technique based on language definitions. In: Proceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012). Vol. 7841 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 248–264.

Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N., 2009. Declarative Debugging of Rewriting Logic Specifications. In: Proceedings of the 19th International Workshop on Algebraic Development Techniques (WADT 2008). Vol. 5486 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 308–325.

Riesco, A., Verdejo, A., Martí-Oliet, N., 2010. Declarative Debugging of Missing Answers for Maude. In: Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010). Vol. 6 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 277–294.

Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R., 2012. Declarative debugging of rewriting logic specifications. The Journal of Logic and Algebraic Programming 81 (7–8), 851–897.

Sabelfeld, A., Myers, A., 2003. Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications 21 (1), 5–19.

Slagle, J., 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. Journal of the ACM (JACM) 21 (4), 622–642.

TeReSe, 2003. Term Rewriting Systems. Cambridge University Press.

Tip, F., 1995. A Survey of Program Slicing Techniques. Journal of Programming Languages 3 (3), 121–189.

Van Deursen, A., Klint, P., Tip, F., 1993. Origin Tracking. Journal of Symbolic Computation 15 (5–6), 523–545.