Inspecting Rewriting Logic Computations (in a parametric and stepwise way) *

M. Alpuente[1], D. Ballis[2], F. Frechina[1], and J. Sapiña[1]

[1] DSIC-ELP, Universitat Politècnica de València,
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain,
[2] CLIP Lab, Technical University of Madrid,
E-28660, Boadilla del Monte, Madrid, Spain,

Abstract. Trace inspection is concerned with techniques that allow the trace content to be searched for specific components. This paper presents a rich and highly dynamic, parameterized technique for the trace inspection of Rewriting Logic theories that allows the non-deterministic execution of a given unconditional rewrite theory to be followed up in different ways. Using this technique, an analyst can browse, slice, filter, or search the traces as they come to life during the program execution. Starting from a selected state in the computation tree, 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 facilitate the dynamic detection of control and data dependencies across the computation tree. Our methodology, which is implemented in the Anima graphical tool, allows users to capture the impact of a given criterion thereby facilitating the detection of improper program behaviors.

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 rely on a set of visualization techniques that facilitate the exploration of the trace content. Common capabilities of these tools include stepping the program execution while searching for particular components and having the option to simplify the traces by hiding some specific contents.

*This work has been partially supported by the EU (FEDER), 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.

Program animation or *stepping* refers to the very common debugging technique of executing code one step at a time, allowing the user to inspect the program state and related data before and after the execution step. This allows the user to evaluate the effects of a given statement or instruction in isolation and thereby gain insight into the program behavior (or misbehavior). Nearly all modern IDEs, debuggers, and testing tools currently support this mode of execution optionally, where animation is achieved either by forcing execution breakpoints, code instrumentation, or instruction simulation.

Rewriting Logic (RWL) is a very general *logical* and *semantic framework*, which is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [**?**] and Web systems [**?**,**?**]). RWL is efficiently implemented in the high-performance system Maude [**?**]. 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 [**?**,**?**,**?**,**?**], but to the best of our knowledge, no versatile program animator or trace inspection tool for RWL/Maude 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 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 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 reduces 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 any RWL-based tool that manipulates unconditional RWL theories. Our formulation takes into account the precise way in which Maude mechanizes the

equational rewriting process modulo B, where B may contain any combination of associativity, commutativity, and unity axioms for different binary operators, and revisits all those rewrite steps in an informed, fine-grained way where each small step corresponds to the application of an equation, equational axiom, or rule. This allows us to explain the input 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 that provides a skillful and highly dynamic interface for the dynamic analysis of RWL computations.

Related Work. Program animators have existed since the early years of programming. Although several steppers have been implemented in the functional programming community (see [**?**] 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 [**?**], which is included in the DrScheme programming environment. The stepper reduces Scheme programs to values (according to the reduction semantics of Scheme) and is useful for explaining the semantics of linguistic facilities and for studying the behavior of small programs. In order to discover all of the steps that occur during the program evaluation, the stepper rewrites (or "instruments") the code, which is in contrast to our technique which does not rely on program instrumentation.

In [**?**,**?**], an incremental, backward trace slicer was presented that generates a trace slice of an execution trace $\mathcal{T}$ by tracing back a set of symbols of interest along (an instrumented version of) $\mathcal{T}$, while data that are not required to produce the target symbols are simply removed. This can be very helpful in debugging since any information that is not strictly needed to deliver a critical part of the result is discarded, which helps answer the question of "what program components might effect a selected computation". However, for the dual problem of "what program components might be effected by a selected computation", a kind of forward expansion is needed (which has been overlooked to date in RWL research).

Plan of the paper. After some preliminaries in Section **??** that describe basic notions of RWL, Section **??** summarizes the rewriting modulo equational theories defined in Maude and provides a convenient trace instrumentation technique that facilitates the stepwise inspection of Maude computations. Section **??** formalizes trace inspection as a semantics-based procedure that is parameterized by the criterion for the inspection. Section **??** formalizes 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 is suitable for analyzing complex, textually-large system computations by filtering out the irrelevant data that do not derive from some selected terms of interest. The Anima tool is described in Section **??**, and Section **??** concludes.

2 Preliminaries

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

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

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Positions are ordered by the prefix ordering; that is, given the positions w_1 and w_2, $w_1 \leq w_2$ if there exists a position u such that $w_1.u = w_2$.

Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t_{|w}$, we denote the *subterm* of t at position w, and $t[s]_w$ specifies the result of *replacing the subterm* $t_{|w}$ by the term s.

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 id 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:

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

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, 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.

A labelled *equation* (or simply *equation*) is an expression of the form $[l] : \lambda = \rho$, where $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, $Var(\rho) \subseteq Var(\lambda)$, and l is a label, i.e., a name that identifies the equation. A labelled *rewrite* rule (or simply *rewrite* rule) is an expression of the form $[l] : \lambda \Rightarrow \rho$, where $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, $Var(\rho) \subseteq Var(\lambda)$, and l is a label. When no confusion can arise, rule and equation labels are often omitted. The term λ (resp., ρ) is called *left-hand side* (resp. *right-hand side*) of the rule $\lambda \Rightarrow \rho$ (resp. equation $\lambda = \rho$).

A *Term Rewriting System* (TRS for short) R is a finite set of rewrite rules. We formalize the rewrite relation $\rightarrow_R$ w.r.t. a TRS R as follows. A rewrite step is the application of a rewrite rule to a term t that replaces a *redex* (reducible expression) of t by its contracted version, or *contractum*. Formally, a term t *rewrites* to a term t' (in symbols $t \overset{r,\sigma,w}{\rightarrow}_R t'$) iff there exists a rewrite rule $[r]$:

$(\lambda \Rightarrow \rho) \in R$, a substitution σ, and a position w of t such that the redex $t_{|w} = \lambda\sigma$ and $t' = t[\rho\sigma]_w$.

3 Rewriting Modulo Equational Theories

Roughly speaking, a rewriting logic theory [20] seamlessly combines a term rewriting system 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 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 (Σ, E), where Σ is an order-sorted signature, $E = \Delta \cup B$ with Δ a collection of (oriented) equations, and B a collection of equational axioms (i.e., algebraic laws such as associativity, commutativity, and unity) that can be associated with any binary operator of Σ[1]. The equational theory (Σ, E) induces a congruence relation on the term algebra $\tau(\Sigma, \mathcal{V})$, which is denoted by $=_E$. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where $(\Sigma, \Delta \cup B)$ is an order-sorted equational theory, and R is a TRS.

Example 1. The following rewrite theory, encoded in Maude, specifies a buggy version of the fault-tolerant client-server communication protocol of [**?**].

```
mod CLIENT-SERVER-TRANSF is inc NAT  .

  sorts Content State Msg Cli Serv Host
   Data CliName ServName Question Answer .

   subsorts Msg Cli Serv < State .
   subsorts CliName ServName < Host .
   subsorts Nat < Question Answer < Data .

   ops Srv-A Srv-B : -> ServName .
   ops Cli-A Cli-B : -> CliName .
   op null : -> State .
   op _&_ : State State -> State [assoc comm
                                  id: null] .
   op _<-_ : Host Content -> Msg .
   op {_,_} : Host Data -> Content .
   op [_,_,_,_] : CliName ServName
                  Question Answer -> Cli .
   op na : -> Answer .
   op [_] : ServName -> Serv .
   op f : ServName CliName Question -> Answer .

   var C S H : Host .
   var Q : Question .
   var A : Answer .
   var D : Data .
   var CNT : Content .

   eq [inc] : f(S, C, Q) = (Q + 1) .

   rl [req]   : [C, S, Q, na] =>
                [C, S, Q, na] &
                 S <- {C, Q} .
   rl [reply] : S <- {C, Q} & [S] =>
                [S] &
                 C <- {S, f(S, C, Q)} .
   rl [rec]   : C <- {S, D} &
                [C, S, Q, A] =>
                [C, S, Q, A] .
   rl [dupl]  : (H <- CNT) =>
                (H <- CNT) & (H <- CNT) .
   rl [loss]  : (H <- CNT) => null .
 endm
```

The specification models an environment where several clients and servers interact. Each server can serve many clients. However, for the sake of simplicity, we assume that each client communicates with a single server.

The names of clients and servers belong to the sorts `CliName` and `ServName`, respectively. Clients are represented as 4-tuples of the form `[C, S, Q, A]`, where `C` is the client's name, `S` is the name of the server it wants to communicate with,

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

`Q` is a natural number that identifies a client request, and `D` is either a natural number that represents the server response, or the constant value `na` (not available) when the response has not yet been received. Servers are stateless and are represented as structures `[S]`, with `S` being the server's name. All messages are represented as pairs of the form `H <- CNT`, where `H` is either the client or server host name, and `CNT` stands for the message contents. These contents are pairs `{H,D}`, with `H` being the host's name and `D` being a data value that represents either a request or a response.

The server `S` uses a function `f` (only known to the server itself) that takes a question `Q` from client `C` as input. This function is defined by means of the equation `inc`, which specifies that the call $\texttt{f}(\texttt{S},\texttt{C},\texttt{Q})$ computes $\texttt{Q} + \texttt{1}$.

Program states are formalized as a soup (multiset) of clients, servers, and messages, whereas the system behavior is formalized through five rewrite rules that model a faulty communication environment in which messages can arrive out of order, can be duplicated, and can be lost. Specifically, the rule `req` allows a client `C` to send a message with request `Q` to the server `S`. The rule `reply` lets the server `S` consume the client request `Q` and send a response message that is computed by means of the function `f`. The rule `rec` specifies the client reception of a server response `D` that should be stored in the client data structure. However, the right-hand side $[\texttt{C},\texttt{S},\texttt{Q},\texttt{A}]$ of the rule `rec` includes an intentional, barely perceptible bug that does not let the client structure be correctly updated with the incoming response `D`. The correct right-hand side should be $[\texttt{C},\texttt{S},\texttt{Q},\texttt{D}]$. Finally, the rules `dupl` and `loss` model the faulty environment and have the obvious meaning: messages can either be duplicated or lost.

Given a rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual rewrite relation on terms $\rightarrow_R$ [**?**] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [**?**]; that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Hence the rewrite relation $\rightarrow_{R/E}$ is defined as $=_E \circ \rightarrow_R \circ =_E$. 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 exploration technique formalized in this work is formulated by considering the precise way in which Maude proves the rewrite steps modulo an equational theory $E = \Delta \cup B$ (see Section 5.2 in [**?**]). Actually, the Maude interpreter implements rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$. These allow rewrite rules and equations to be intermixed in the rewriting process by simply using an algorithm of matching modulo 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 normalized 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 mod-

ulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [**?**].

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

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 [**?**]:

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 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 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.

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

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 $s \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$ t. We define $m\mathcal{S}(s)$ as the set of all such non-deterministic Maude steps stemming from s.

3.1 Instrumented Computations

In this section, we introduce an auxiliary technique for instrumenting computation traces. 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 despite the fact that terms are rewritten modulo 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 [**?**]. 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 [**?**].

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 identity element of U is not included in the U-normal form, and variables under a U symbol can always be assigned the identity element through U-matching [**?**].

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, in a Maude step, all terms in the sequence are 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.

Example 2. 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 $f(b, f(f(b, a), c)) \stackrel{\texttt{toACnf}}{\longrightarrow} f^*(a, b^2, c) \stackrel{\texttt{fromACnf}}{\longrightarrow} f(f(b, a), f(c, b))$, where 1) the first step corresponds to a term transformation that obtains the AC-normal form

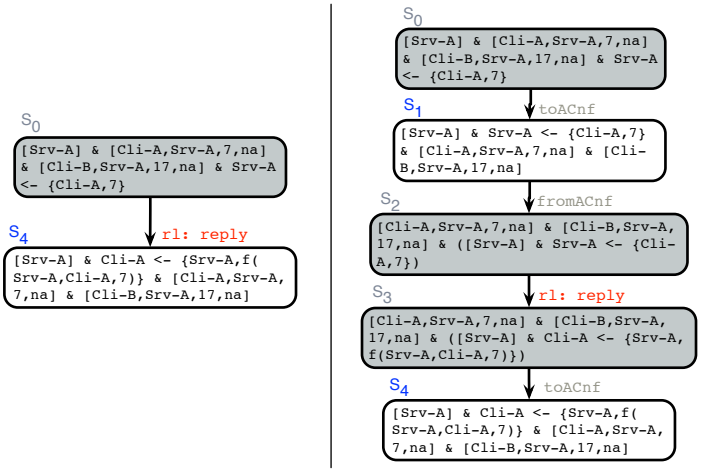


Figure 1. A rewrite step and its instrumented version

$f^*(a, b^2, c)$, and 2) the second step corresponds to the inverse, 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 $f^*(a, b^2, c) \xrightarrow{\texttt{fromACnf}} f(f(b, c), f(a, b))$ that actually delivers the second AC-matcher $\{x/b, y/c, z/a\}$.

Obviously, in our implementation, rewriting modulo B proceeds by using the standard form of B-matching on B-normal forms supported by Maude, where B-normalization is applied both to the states and to the (left-hand sides and right-hand sides) of the rules. The artifice described above is only a means to reveal the term transformations of subterms forced by the step so that any position can be properly traced across rewriting steps. Let us see an example.

Example 3. Consider the rewrite theory of Example **??** together with the rewrite step and corresponding instrumentation shown in Figure **??**, where B-normalized nodes are represented in white, whereas nodes not in B-normal form are shown shaded in grey. The instrumented version of the rewrite step reveals that the normalized rule[2]

```
rl [reply] : [S] & S <- {C, Q}  => [S] & C <- {S, f(S, C, Q)} .
```

is not actually applied into the term s_0, but rather into a B-equivalent term s_2 that is chosen to syntactically match the left-hand side of the applied rule. As a result, all the information we collect from the application of the rule (e.g., the

[2] Note that, in this specific case, the B-normalization of the `reply` rule simply consists of a reordering of arguments in the left-hand side of the rule. Given any program rule, when no confusion can arise we always use the same label for the original rule and for the B-normalized version of the rule that is internally used by Maude.

position where the rule was applied) corresponds to the s_1, s_2, and s_3 states, which are omitted in the non-instrumented version of the rewrite step.

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

Example 4. Consider the rewrite theory of Example **??** together with the following computation $\mathcal{C}$ that consists of a single Maude step (note that the last term is normalized):

$$
\begin{aligned}
\mathcal{C} = \texttt{[Srv-A] \& Cli-A <- \{Srv-A, f(Srv-A, Cli-A, 7)\}} \\
\texttt{\& [Cli-A, Srv-A, 7, na]} \overset{\texttt{inc}}{\longrightarrow}_{\Delta,B} \\
\texttt{[Srv-A] \& Cli-A <- \{Srv-A, 8\} \& [Cli-A, Srv-A, 7, na]} \overset{\texttt{rec}}{\longrightarrow}_{R,B} \\
\texttt{[Srv-A] \& [Cli-A, Srv-A, 7 , na]}
\end{aligned}
$$

The corresponding instrumented computation $\mathcal{T}$, produced by *instrument*($\mathcal{C}$), is given by suitably parenthesizing and reordering the arguments of the second term by applying ACU-matching transformations for the operator _&_.

These internal transformations allow the `rec` rule to be applied by syntactically matching the third term of $\mathcal{T}$ within its left-hand side.

$$
\begin{aligned}
\mathcal{T} = \texttt{[Srv-A] \& Cli-A <- \{Srv-A, f(Srv-A, Cli-A, 7)\}} \\
\texttt{\& [Cli-A, Srv-A, 7, na]} \overset{\texttt{inc}}{\longrightarrow}_{\Delta} \\
\texttt{[Srv-A] \& Cli-A <- \{Srv-A, 7+1\}} \\
\texttt{\& [Cli-A, Srv-A, 7, na]} \overset{\texttt{builtIn}(+)}{\longrightarrow}_{\Delta} \\
\texttt{[Srv-A] \& Cli-A <- \{Srv-A, 8\} \& [Cli-A, Srv-A, 7, na]} \overset{\texttt{fromACUnf}}{\longrightarrow}_{B} \\
\texttt{[Srv-A] \& (Cli-A <- \{Srv-A, 8\} \& [Cli-A, Srv-A, 7, na])} \overset{\texttt{rec}}{\longrightarrow}_{R} \\
\texttt{[Srv-A] \& [Cli-A, Srv-A, 7, na]}
\end{aligned}
$$

The second rewrite step of the instrumented trace is simply proven with the bogus rule:

```
rl [fromACUnf] : [Srv-A] & Cli-A <- {Srv-A, 8}
                         & [Cli-A, Srv-A, 7, na] =>
                 [Srv-A] & (Cli-A <- {Srv-A, 8}
                         & [Cli-A, Srv-A, 7, na]) .
```

In order to improve readability, we omit B-matching transformations and built-in evaluations when displaying Maude steps (unless explicitly stated otherwise). This is consistent with the strategy adopted by Maude and is the default

option in our tool. As described in Section **??**, by using the tool Anima, the user can 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 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*[3], $\mathcal{T}_{\mathcal{R}}(s)$. 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 5. Consider the rewrite theory of Example **??** together with the initial term `[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]`. In this case, the computation tree consists of several infinite computations that start from the considered initial term and model interactions between clients `Cli-A`, `Cli-B` and server `Srv-A`. A fragment of the computation tree is depicted in Figure **??** where we only display the equations and 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. Also for simplicity, note that we merge the two edges leading from s_1 to the same node s_4 with the rules req and dup, respectively.

Note that 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 **??**. 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 algebraic axiom applications 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 slicing-based exploration technique that allows the user to incrementally generate and inspect a portion 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.

[3] 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.

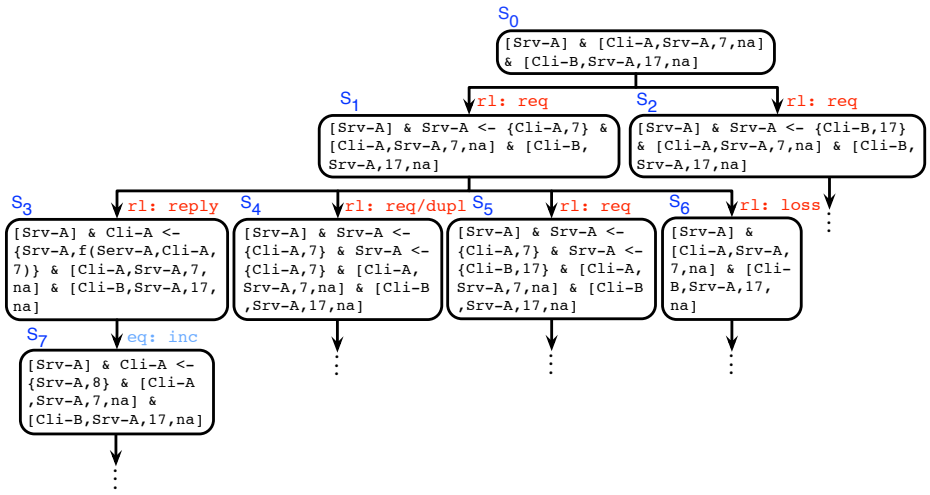


Figure 2. Computation tree

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 $i = 0, 1, 2, \ldots$. 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, a term slice records just the information the user wants to observe of a given term.

Example 6. Consider the client-server specification of Example **??**. Then, the term slice `[Cli-A, Srv-A,` $\bullet_1$`,` $\bullet_2$`]` represents any request from client `Cli-A` to communicate with server `Srv-A` where the request and response identification numbers are irrelevant. For this term slice, the set of meaningful positions is $\{\Lambda, 1, 2\}$.

The next auxiliary definition formalizes the function $Tslice(t, P)$ that 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 $fresh^\bullet$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort that is distinct from any previously generated variable $\bullet_j$.

Definition 1 (Term Slice). *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. Then, the term slice $Tslice(t, P)$ of t w.r.t. P is computed as follows.*

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

Figure 3. The inference rule frag of the transition system $(\mathit{Conf}, \Longrightarrow)$.

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

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

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 maximal subterm $t_{|p}$, with $p \notin P$, is replaced by means of a freshly generated $\bullet$-variable.

Example 7. Let $t = d(f(g(a, h(b)), c), a)$ be a term, and let $P = \{1.1,\ 1.2\}$ be a set of positions of t. By applying Definition **??**, we get the term slice $t^\bullet = \mathit{Tslice}(t, P) = d(f(g(\bullet_1, \bullet_2), c), \bullet_3)$ and the set of meaningful positions $\mathcal{MP}os(t^\bullet) = \{\Lambda, 1, 1.1, 1.2\}$.

Definition 2 (Inspection criterion). *An* inspection criterion *is a function* $\mathcal{I}(s^\bullet, s \to_K t)$ *that, given a* $\to_K$*-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 **??**. We assume that the special value fail is returned by the inspection criterion whenever no slice $t^\bullet$ can be computed by $\mathcal{I}$. Actually, for any sensible criterion $\mathcal{I}$, $\mathcal{I}(\bullet, s \ \to_K t) = \mathsf{fail}$ (i.e., no meaningful result can be derived when no relevant information is considered).

Given the instrumented computation $\mathcal{T} = (s_0 \ \to_K \ s_1 \ldots \to_K \ s_n)$, with $n \geq 1$, an *instrumented computation slice* of $\mathcal{T}$ w.r.t. the inspection criterion $\mathcal{I}$ is the sequence $\mathcal{T}^\bullet_\mathcal{I} = (s^\bullet_0 \bullet\!\!\to s^\bullet_1 \bullet\!\!\to \ldots \bullet\!\!\to s^\bullet_n)$ that can be 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 $(\mathit{Conf}, \Longrightarrow)$ [**?**] where

- *Conf* is a set of *configurations* of the form $\langle \mathcal{T}, \mathcal{F}^\bullet \rangle$, where $\mathcal{T}$ is a an instrumented computation and $\mathcal{F}^\bullet$ is an instrumented computation slice of a prefix of $\mathcal{T}$;
- 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 **??**. By $\Longrightarrow^*$, we denote the usual transitive and reflexive closure of the relation $\Longrightarrow$.

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 3 (Computation slice). *Given the instrumented computation* $\mathcal{T} = (s_0 \to_K s_1 \to_K \ldots \to_K s_n)$*, 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* $s_0^\bullet$ *of* s_0 *is defined by the function* $Cslice(s_0^\bullet, \mathcal{T}, \mathcal{I})$ *which is defined as follows.*

$$Cslice(s_0^\bullet, \mathcal{T}, \mathcal{I}) = \textbf{if } \langle \mathcal{T}, s_0^\bullet \rangle \Longrightarrow^* \langle \varepsilon, \mathcal{T}^\bullet \rangle \textbf{ then } \mathcal{T}^\bullet \textbf{ else } \mathsf{fail}$$

where ε *denotes the empty computation. Note that the second component* $s_0^\bullet$ *of the initial configuration* $\langle \mathcal{T}, s_0^\bullet \rangle$ *matches the sequence* $S^\bullet \bullet\!\!\to^* U^\bullet$ *in rule frag by taking* $s_0^\bullet$ *for* $U^\bullet$ *and considering a sequence* $S^\bullet \bullet\!\!\to^* U^\bullet$ *consisting of zero steps.*

4.2 Instrumented Computation Tree Slices

Instrumented computation tree slices are formally defined as follows.

Definition 4 (Instrumented Computation Tree Slice). *Let* $\mathcal{T}_\mathcal{R}^+(s_0)$ *be an instrumented computation tree for the term* s_0 *in the 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.

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** $\mathcal{M}^\bullet \neq$ fail **then** $\mathcal{A} = \mathcal{A} \cup \{\mathcal{M}^\bullet\}$
5. **end**
6. **return** $\mathcal{A}$

endf

Figure 4. The one-step *expand* function.

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 **??** 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 then added to the set of arcs $\mathcal{A}$.

The overall construction methodology for instrumented computation tree slices is specified by the function *explore*, defined in Figure **??**. 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 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 *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 *addPaths*$(\mathcal{T}^+_\mathcal{R}(s_0^\bullet), s^\bullet, \mathit{expand}(s, s^\bullet, \mathcal{R}, \mathcal{I}))$. 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 *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 Particularizing the Exploration

The methodology given in Section **??** provides a generic scheme for the exploration of (instrumented) computation trees w.r.t. a given inspection criterion $\mathcal{I}$

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

endf

Figure 5. The interactive *explore* function.

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 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.

5.1 Interactive Stepper

Given an instrumented computation tree $\mathcal{T}^+_{\mathcal{R}}(s_0)$ for an initial term s_0 and a rewrite theory $\mathcal{R}$, the stepwise inspection of the computation tree can be directly implemented by instantiating the exploration scheme of Section **??** with the basic inspection criterion $\mathcal{I}_{step}(s, s \overset{r,\sigma,w}{\rightarrow}_K t) = t$ which simply returns the reduced term t of the rewrite step $s \overset{r,\sigma,w}{\rightarrow}_K t$.

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 $explore(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I}_{step})$ 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 8. Consider the rewrite theory $\mathcal{R}$ in Example **??** and the computation tree in Example **??**. Assume the user starts the exploration by calling $explore(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I}_{step})$, 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 **??**.

Now, the user can either quit or carry on with the exploration of nodes s_3 and s_5, which would result in the instrumented version of the tree fragment that is shown in Figure **??**.

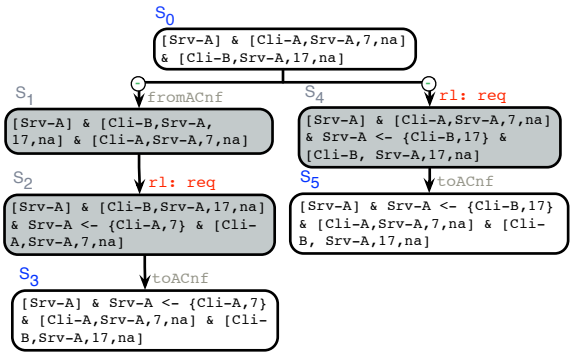


Figure 6. Inspection of the state s_0 w.r.t. $\mathcal{I}_{step}$

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 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 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 work with state information of interest, thereby producing more compact and focused representations of the visited slices of the (instrumented) computation trees.

Example 9. Consider the computation tree of Example **??** whose initial term is s_0 = `[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]`. Let $s_0^\bullet$ = (•$_1$ `&` `[Cli-A,Srv-A,7,na]` `&` •$_2$) be a term slice of s_0 where only client `Cli-A` data structure is considered of interest. Assume that the inspection criterion $\mathcal{I}_{pstep}$ is used to generate computation tree slice fragments. The computation tree slice fragment shown in Figure **??** is obtained by first expanding the node $s_0^\bullet$ into $s_1^\bullet$, and then the node $s_1^\bullet$ into $s_2^\bullet$, $s_3^\bullet$. The expanded nodes have been highlighted in the figure. Note that the adopted partial stepping strategy allows a simplified view of (a part of) the considered computation tree to be constructed. Specifically, the generated computation tree slice fragment isolates client `Cli-A`'s behavior. More precisely, given the input encoded in the initial term slice $s_0^\bullet$, the

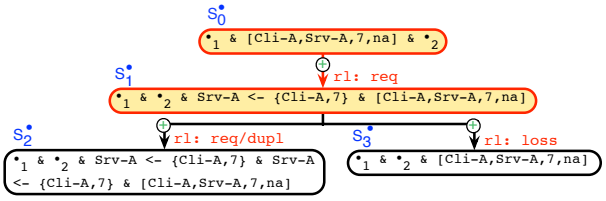


Figure 7. Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{pstep}$

computation can evolve by only applying either the rule `req` to the `Cli-A` data structure, or the rules `dupl` and `loss` to `Cli-A`'s request messages.

In other words, this amounts to saying that a client-server protocol interaction cannot be successfully carried out when the input term does not specify a sever data structure (in this specific case, `[Srv-A]` should be included in $s_0^\bullet$), since its presence is essential to fire the `reply` rule that is in charge of producing server responses.

5.3 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 $s_0^\bullet$ 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 $s_0^\bullet$, while irrelevant data are simply pruned away.

In the following, we define an inspection criterion $\mathcal{I}_{slice}$ that implements the forward slicing for a single rewrite step. Given a rewrite step $\mu = (s \overset{r,\sigma,w}{\rightarrow}_K t)$ (with $r = \lambda \Rightarrow \rho$) and a term slice $s^\bullet$ of the term s, it delivers the term slice $t^\bullet$ that results from "rewriting" $s^\bullet$ at position w with the rule r and a suitable substitution that abstracts any irrelevant information of the computed substitution σ with $\bullet$-variables. A precise formalization of the inspection criterion $\mathcal{I}_{slice}$ is provided by the algorithm in Figure **??**.

Note that, by adopting the inspection criterion $\mathcal{I}_{slice}$, the exploration scheme of Section **??** 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 $s_0^\bullet$ of the initial term s_0, any computation slice $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet \ldots \bullet\!\!\rightarrow s_n^\bullet$ in the tree $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$, which is computed by the *explore* function, is the sliced counterpart of an instrumented computation $s_0 \rightarrow s_1 \ldots \rightarrow s_n$ (w.r.t. the term slice $s_0^\bullet$) 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 μ occurs at a position w that is not a meaningful position of $s^\bullet$

function $\mathcal{I}_{slice}(s^\bullet, s \overset{\lambda \Rightarrow \rho, \sigma, w}{\to_K} t)$
1. **if** $w \in \mathcal{MP}os(s^\bullet)$ **then**
2. $\quad \theta = \{x/fresh^\bullet \mid x \in Var(\lambda)\}$
3. $\quad \lambda^\bullet = Tslice(\lambda, \mathcal{MP}os(s^\bullet_{|w}) \cap \mathcal{P}os(\lambda))$
4. $\quad \psi_\lambda = \langle\!|\theta, match_{\lambda^\bullet}(s^\bullet_{|w})|\!\rangle$
5. $\quad t^\bullet = s^\bullet[\rho\psi_\lambda]_w$
6. **else**
7. $\quad t^\bullet = \mathsf{fail}$
8. **fi**
9. **return** $t^\bullet$

endf

Figure 8. Inspection criterion that models the forward slicing of a rewrite step

(in symbols, $w \notin \mathcal{MP}os(s^\bullet)$), trivially μ does not contribute to producing the meaningful symbols of $t^\bullet$. This amounts to saying that no relevant information descends from the term slice $s^\bullet$ and, hence, the function returns the fail value.

On the other hand, when $w \in \mathcal{MP}os(s^\bullet)$, the computation of $t^\bullet$ involves a more in-depth analysis of the rewrite step, which is based on a refinement process that allows the descendants of $s^\bullet$ in $t^\bullet$ to be computed.

The following definition is auxiliary and is used to update (override) a substitution σ_1 with the substitution σ_2, where both σ_1 and σ_2 may contain $\bullet$-variables.

Definition 5 (substitution update). *Let σ_1 and σ_2 be two substitutions,. The* update *of σ_1 w.r.t. σ_2 (in symbols $\langle\!|\sigma_1, \sigma_2|\!\rangle$) is defined by $\langle\!|\sigma_1, \sigma_2|\!\rangle = \sigma_{\restriction Dom(\sigma_1)}$, where the substitution σ is given by*

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

The main idea behind the operator $\langle\!|\_, \_|\!\rangle$ is that, in order to compute a rewrite step from the term slice $s^\bullet$ using the rule r, all variables in r are naïvely assumed to be initially bound to $\bullet$-variables that model irrelevant data, and the bindings are incrementally updated as we apply the rule r.

More specifically, given the rewrite step $\mu : s \overset{r, \sigma, w}{\to} t$, with $r = \lambda \Rightarrow \rho$, and the term slice $s^\bullet$, we initially define the substitution $\theta = \{x/fresh^\bullet \mid x \in Var(\lambda)\}$ that binds each variable in $\lambda \Rightarrow \rho$ to a fresh $\bullet$-variable. This corresponds to assuming that all the information in μ, which is introduced by the substitution σ, can be marked as irrelevant. Then, θ is refined as follows.

We first compute the term slice $\lambda^\bullet = Tslice(\lambda, \mathcal{MP}os(s^\bullet_{|w}) \cap \mathcal{P}os(\lambda))$ that filters the meaningful symbols of the left-hand side λ of the rule r w.r.t. the set of meaningful positions of $s^\bullet_{|w}$. Then, by matching $s^\bullet_{|w}$ into $\lambda^\bullet$, we generate a matcher $match_{\lambda^\bullet}(s^\bullet_{|w})$ that extracts the meaningful symbols from $s^\bullet_{|w}$. Such a matcher is then used to compute ψ_λ, which is an update of θ w.r.t. $match_{\lambda^\bullet}(s^\bullet_{|w})$ containing the meaningful information to be propagated across the rewrite step. Finally, the term slice $t^\bullet$ is computed from $s^\bullet$ by replacing its subterm at position

w with the instance $\rho\psi_\lambda$ of the right-hand side of the applied rule r. This way, we can transfer all the relevant information marked in $s^\bullet$ into the slice of the resulting term $t^\bullet$.

Example 10. Consider the rewrite theory in Example ?? together with the following rewrite step

$$s \overset{\texttt{req}}{\to} t : \texttt{[Srv-A] \& [Cli-A,Srv-A,7,na]} \overset{\texttt{req}}{\to} \texttt{[Srv-A] \& Srv-A <- \{Cli-A,7\} \& [Cli-A,Srv-A,7,na]}$$

that applies (at position $w = 2$) the rule `req:` $\lambda \Rightarrow \rho$, with $\lambda =$`[C,S,Q,na]` and $\rho =$ `[C, S, Q, na] & S <-` $\{$`C, Q`$\}$.

Let $s^\bullet =$ $\bullet_1$ `& [Cli-A,`$\bullet_2$`,7,` $\bullet_3$ `]` be a term slice of s. The execution of the inspection criterion $\mathcal{I}_{slice}(s^\bullet, s \overset{\texttt{req}}{\to} t)$ that computes a term slice $t^\bullet$ of t proceeds as follows.

First, the substitution θ is initialized to $\{$`C/`$\bullet_4$, `S/`$\bullet_5$, `Q/`$\bullet_6\}$ and the slice $\lambda^\bullet$ of λ is computed w.r.t. the meaningful positions of $s^\bullet_{|\Lambda.2}$ that also appear in λ. Specifically, $\lambda^\bullet = \mathit{Tslice}(\lambda, \{\Lambda,\ 1,\ 3\}) =$ `[C,`$\bullet_7$`,Q,`$\bullet_8$`]`. Then, the update ψ_λ of θ is calculated. More precisely, $\mathit{match}_{\lambda^\bullet}(s^\bullet_{|w}) = \mathit{match}_{\texttt{[C,}\bullet_7\texttt{,Q,}\bullet_8\texttt{]}}(\texttt{[Cli-A,}\bullet_2\texttt{,7,}\bullet_3\texttt{]})$ $=$ $\{$`C/Cli-A`, $\bullet_7/\bullet_2$, `Q/7`, $\bullet_8/\bullet_3\}$ and $\psi_\lambda = \langle\!|\theta, \mathit{match}_{\lambda^\bullet}(s^\bullet_{|w})|\!\rangle = \{$`C/Cli-A`, `S/`$\bullet_5$, `Q/7`$\}$. Roughly speaking, the computed update ψ_λ refines θ by replacing the uninformed bindings `C/`$\bullet_4$ and `Q/`$\bullet_6$ with `C/Cli-A` and `Q/7`, respectively. Finally, $\mathcal{I}_{slice}(s^\bullet, s \overset{\texttt{req}}{\to} t)$ returns the term slice $t^\bullet = s^\bullet[\rho\psi_\lambda]_2 =$ $\bullet_1$ `& [Cli-A,` $\bullet_5$ `,7,na] &` $\bullet_5$ `<-`$\{$`Cli-A,7`$\}$.

The following example describes the interactive construction process of a fragment of an instrumented computation tree slice based on the $\mathcal{I}_{slice}$ criterion. To improve its readability, we omit the transformation steps that are required to mimick the behavior of the Maude B-matching algorithm. The example also demonstrates how forward trace slicing can be fruitfully employed to debug RWL specifications.

Example 11. Consider the computation tree of Example ?? whose initial term is $s_0 =$ `[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]`. Let $s_0^\bullet =$ ($\bullet_1$ `& [Cli-A,`$\bullet_2$`,7,` $\bullet_3$ `] &` $\bullet_4$) be a term slice of s_0 where only request `7` of client `Cli-A` is considered of interest. We get the computation tree slice fragment shown in Figure ?? by first expanding (w.r.t. the inspection criterion $\mathcal{I}_{slice}$) the node $s_0^\bullet$ into $s_1^\bullet$; the node $s_1^\bullet$ into $s_2^\bullet$, $s_3^\bullet$ (which is automatically normalized to $s_5^\bullet$ using the equation `inc`), and $s_4^\bullet$; and finally the node $s_5^\bullet$ into $\{s_6^\bullet \ldots s_9^\bullet\}$. The branch leading from $s_0^\bullet$ to $s_9^\bullet$ is highlighted.

Note that the intermediate node $s_3^\bullet$ does not have to be expanded since it is an intermediate node generated by the expansion of node $s_1^\bullet$ that is automatically normalized into $s_5^\bullet$. Indeed, the computation slice generated by expanding the node $s_1^\bullet$ is $s_1^\bullet \overset{\texttt{reply}}{\bullet\!\!\to} s_3^\bullet \overset{\texttt{inc}}{\bullet\!\!\to} s_5^\bullet$, which corresponds to the forward slicing of a Maude step from s_1.

The slicing process automatically computes a computation tree slice fragment that represents a partial view of the protocol interactions from client `Cli-A`'s

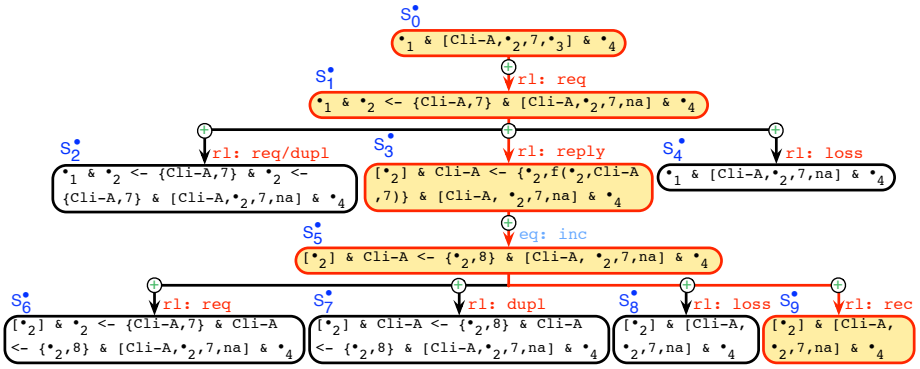


Figure 9. Computation tree slice fragment for $\bullet_1$ `&` `[Cli-A,`$\bullet_2$`,7,` $\bullet_3$ `]` `&` $\bullet_4$ w.r.t. $\mathcal{I}_{slice}$

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, if we observe the highlighted branch in Figure **??**, we can easily detect the wrong behavior of the rule `rec`. Specifically, by inspecting the term slice $s_9^\bullet = ([\bullet_2]$ `&` `[Cli-A,`$\bullet_2$`, 7,na]` `&` $\bullet_4)$, which is generated by an application of the rule `rec`, we immediately realize that response `8` produced in the parent node $s_5^\bullet$ has not been stored in $s_9^\bullet$, which clearly reveals the bug in the applied rule `rec`.

Finally, it is worth noting how the forward trace slicer implemented via the criterion $\mathcal{I}_{slice}$ differs from the partial stepper given at the end of Section **??**. Given a term slice $s^\bullet$ and a rewrite step $s \xrightarrow{r,\sigma,w}_K t$, $\mathcal{I}_{slice}$ always yields a slice $t^\bullet$ when the rewrite step occurs at a meaningful position, whereas the inspection criterion $\mathcal{I}_{pstep}$ encoded in the partial stepper may fail to provide a computed slice $t^\bullet$ when $s^\bullet$ does not rewrite to $t^\bullet$, which allows the user to identify those states that can be reached, from any instance of the sliced input state, by standard rewriting.

Example 12. Consider the computation tree of Example **??** whose initial term is s_0 = `[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]`, and the initial term slice $s_0^\bullet = (\bullet_1$ `&` `[Cli-A,`$\bullet_2$`,7,` $\bullet_3$ `]` `&` $\bullet_4)$ of Example **??**. Then, no expansion of node $s_0^\bullet$ is possible using the inspection criterion $\mathcal{I}_{pstep}$, since the input encoded in $s_0^\bullet$ does not suffice to enable the application of any protocol rule, whereas the forward slicing strategy specified by the criterion $\mathcal{I}_{slice}$ allows the computation tree fragment in Figure **??** to be generated. Nevertheless, tree fragments computed by forward slicing do not generally describe valid computations, that is, computations that can be proven for any instance of the sliced input state.

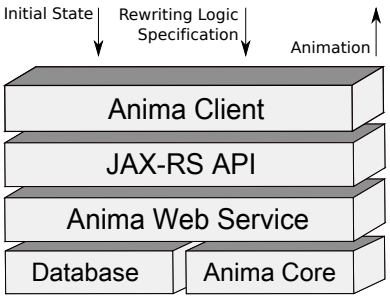


Figure 10. Anima architecture.

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 consists of about 250 Maude function definitions (approximately 2000 lines of source code). 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 **??** 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[4] 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 on our needs.

Figure **??** displays a screenshot that shows the Anima tool at work on the case study that is described in Example **??**. The figure depicts (a fragment of) the computation tree slice for this example program and several capabilities offered by the tool.

These are the main features provided by Anima:

1. *Inspection strategies.* The tool implements the three inspection strategies described in Section **??**. As shown in Figure **??**, the user can select the desired strategy by using the selector provided in the option pane.
2. *Select meaningful symbols for slicing.* State slices can be specified by highlighting with the mouse the state symbols of interest directly on the tree.
3. *Expand/Fold program states.* The user can expand or fold states of the tree by double-clicking or right-clicking on them with the mouse and then selecting

[4] For the sake of efficiency, browsers limit the maximum dimensions of a canvas object (eg., 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.

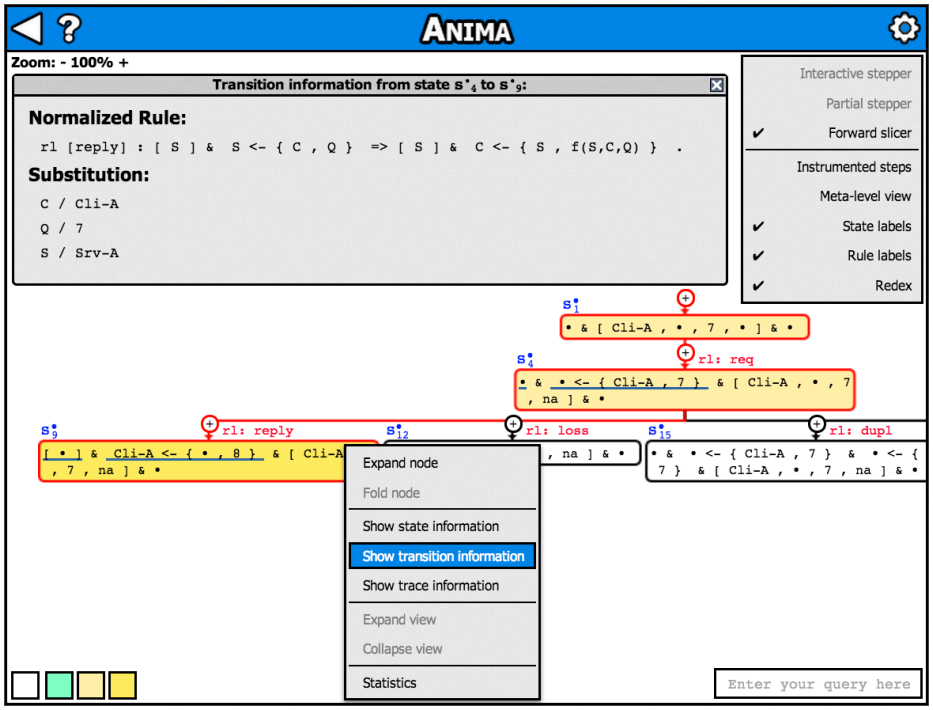


Figure 11. Anima at work.

either the *Expand Node* option or *Fold Node* option that are offered in the contextual menu. For instance, in Figure **??**, a state slice on the frontier of the computed tree slice fragment has been selected and is ready to be expanded through the *Expand Node* option that will add all the possible slices of the Maude steps to the tree starting from the selected node. The whole branch leading from the root to the selected node of the tree is highlighted. 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.

4. *Display instrumented trace.* 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 in 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.
5. *Tree Query mechanism.* The search facility illustrated in Figures **??** and **??** implements a pattern language that allows the selected information of

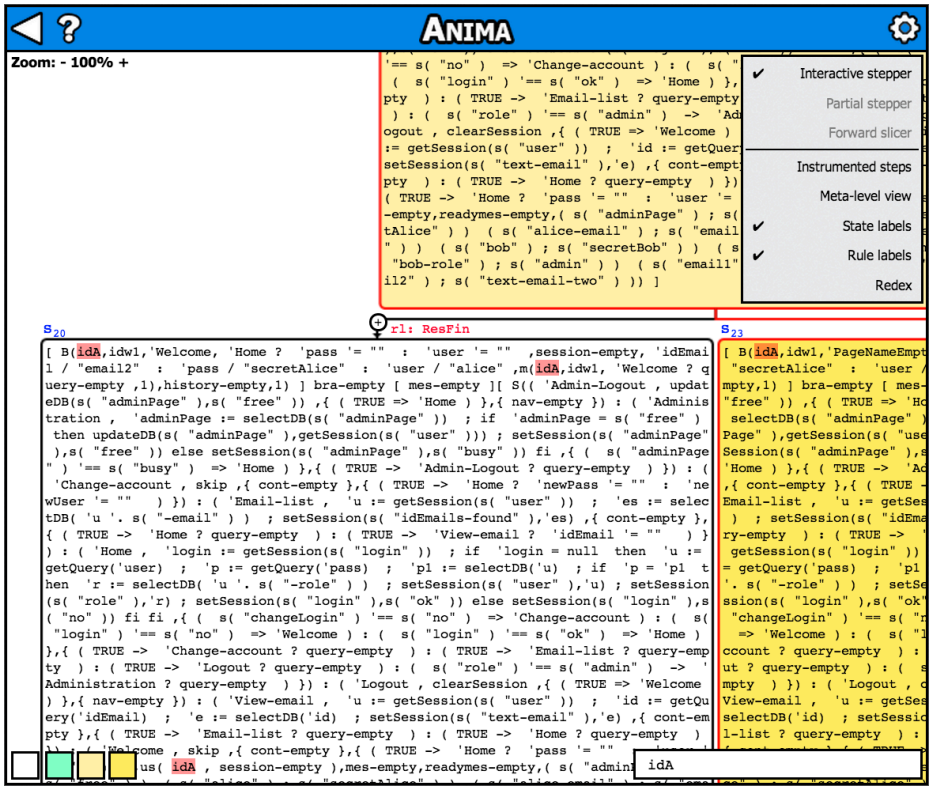
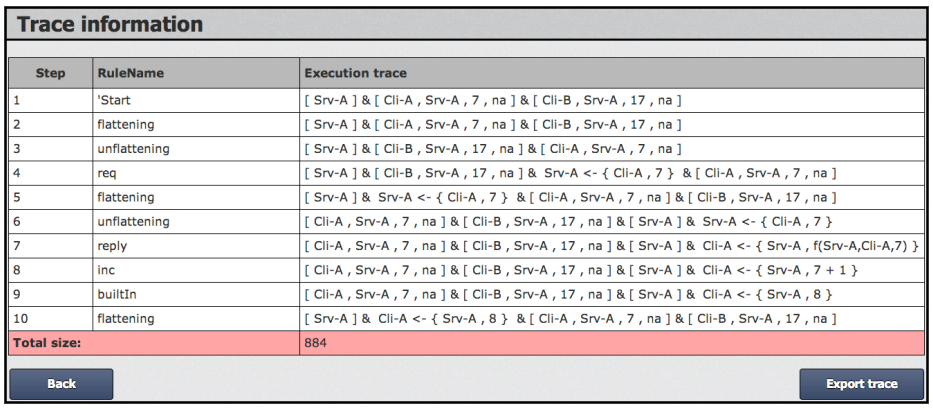


Figure 12. Anima search mechanism.

interest to be searched on 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.

6. *Show 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 cannot highlight in general 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 identity elements, and parenthesised, yielding the B-equivalent terms 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').



Trace information

Step	RuleName	Execution trace
1	'Start	[Srv-A] & [Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na]
2	flattening	[Srv-A] & [Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na]
3	unflattening	[Srv-A] & [Cli-B , Srv-A , 17 , na] & [Cli-A , Srv-A , 7 , na]
4	req	[Srv-A] & [Cli-B , Srv-A , 17 , na] & Srv-A <- { Cli-A , 7 } & [Cli-A , Srv-A , 7 , na]
5	flattening	[Srv-A] & Srv-A <- { Cli-A , 7 } & [Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na]
6	unflattening	[Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na] & [Srv-A] & Srv-A <- { Cli-A , 7 }
7	reply	[Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na] & [Srv-A] & Cli-A <- { Srv-A , f(Srv-A,Cli-A,7) }
8	inc	[Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na] & [Srv-A] & Cli-A <- { Srv-A , 7 + 1 }
9	builtIn	[Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na] & [Srv-A] & Cli-A <- { Srv-A , 8 }
10	flattening	[Srv-A] & Cli-A <- { Srv-A , 8 } & [Cli-A , Srv-A , 7 , na] & [Cli-B , Srv-A , 17 , na]
Total size:		884

Back Export trace

Figure 13. Anima trace information.

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.

7. *Show 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 **??**. 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, for example, *i*Julienne [**?**].
8. *Show 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

The analysis of execution traces plays a fundamental role in many program analysis approaches, such as runtime verification, monitoring, testing, and specification mining. We have presented a parametrized exploration technique that can be applied to the inspection of rewriting logic computations and that can work in different ways. Three instances of the parameterized exploration scheme (an incremental stepper, an incremental partial stepper, and a forward trace slicer)

have been formalized and implemented in the Anima tool, which is a novel program animator for RWL. 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.

As already mentioned, the present version supports the instrumentation of matching modulo associativity, commutativity, and (left-, right- or two-sided) unity. Nevertheless, Anima has an extensible design so that instrumentation for other equational axioms such as idempotency can be easily added in the future.

As future work, we intend to apply our exploration technique to more sophisticated rewrite theories that may include membership axioms as well as conditional rules and equations. Furthermore, we plan to integrate the analysis capabilities of the backward trace slicer *i*Julienne [**?**] in Anima. The idea is to first apply forward trace slicing to a given computation in order to remove all the information that does not affect the observed symbols. This procedure may produce "incorrect" computation slices, that is, computation slices that do not precise all the concrete input data that are required to generate the relevant symbols in the output/final state of the computation slice, as seen in Example **??**. Hence, backward trace slicing might be applied to the generated computation slice to enrich it with new input symbols computed as antecedents of the relevant output with the aim of ensuring the correctness of the slice.

Finally, we envisage to equip Anima with dynamic program slicing techniques to extract the minimal program slice that is needed to generate any selected execution trace of the computation tree.

References

1. M. Alpuente, D. Ballis, M. Baggi, and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In *Proc. PEPM 2010*, pp. 43–52. ACM, 2010.
2. M. Alpuente, D. Ballis, J. Espert, and D. Romero. Model-checking Web Applications with Web-TLR. In *Proc. ATVA 2010*, vol. 6252 of *LNCS*, pp. 341–346. Springer-Verlag, 2010.
3. M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward Trace Slicing for Rewriting Logic Theories. In *Proc. CADE 2011*, vol. 6803 of *LNCS*, pp. 34–48. Springer-Verlag, 2011.
4. M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Slicing-Based Trace Analysis of Rewriting Logic Specifications with *i*Julienne. In *Proc. ESOP 2013*, vol. 7792 of *LNCS*, pp. 121–124. Springer-Verlag, 2013.
5. M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Using Conditional Trace Slicing for improving Maude programs. *Science of Computer Programming*, to appear, 2013.
6. M. Alpuente, D. Ballis, and D. Romero. A Rewriting Logic Approach to the Formal Specification and Verification of Web applications. *Science of Computer Programming*, to appear, 2013.

7. M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In *Proc. CMSB 2009*, vol. 5688 of *LNCS*, pp. 68–82. Springer-Verlag, 2009.
8. R. Bruni and J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.6). Technical report, SRI Int'l Computer Science Laboratory, 2011. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.
10. J. Clements, M. Flatt, and M. Felleisen. Modeling an Algebraic Stepper. In *Proc. ESOP 2001*, vol. 2028 of *LNCS*, pp. 320–334. Springer-Verlag, 2001.
11. F. Durán and J. Meseguer. A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In *Proc. WRLA 2010*, vol. 6381 of *LNCS*, pp. 86–103. Springer-Verlag, 2010.
12. S. Eker. Associative-Commutative Matching via Bipartite Graph Matching. *The Computer Journal*, 38(5):381–399, 1995.
13. S. Eker. Associative-Commutative Rewriting on Large Terms. In *Proc. RTA 2003*, vol. 2706 of *LNCS*, pp. 14–29. Springer-Verlag, 2003.
14. J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. I, pp. 1–112. Oxford University Press, 1992.
15. N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
16. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
17. J. Meseguer. The Temporal Logic of Rewriting: A Gentle Introduction. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of his 65th Birthday*, vol. 5065 of *LNCS*, pp. 354–382. Springer-Verlag, 2008.
18. G. D. Plotkin. The Origins of Structural Operational Semantics. *The Journal of Logic and Algebraic Programming*, 60-61(1):3–15, 2004.
19. A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative Debugging of Rewriting Logic Specifications. In *Proc. WADT 2008*, vol. 5486 of *LNCS*, pp. 308–325. Springer-Verlag, 2009.
20. A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative Debugging of Missing Answers for Maude. In *Proc. RTA 2010*, vol. 6 of *LIPIcs*, pp. 277–294, 2010.
21. TeReSe. *Term Rewriting Systems*. Cambridge University Press, 2003.