Parametric Exploration of Rewriting Logic Computations *

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,
{alpuente,ffrechina,jsapina}@dsic.upv.es
[2] DIMI, Università degli Studi di Udine,
Via delle Scienze 206, 33100 Udine, Italy, demis.ballis@uniud.it

Abstract

This paper presents a parameterized technique for the inspection of Rewriting Logic computations that allows the non-deterministic execution of a given rewrite theory to be followed up in different ways. Starting from a selected state in the computation tree, the exploration is driven by a user-defined, inspection criterion that specifies the exploration mode. By selecting different inspection criteria, one can automatically derive useful debugging facilities such as program steppers and more sophisticated dynamic trace slicers that facilitate the 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, validate input data, and detect improper program behaviors.

1 Introduction

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 [7] and Web systems [2, 6]). RWL is efficiently implemented in the high-performance system Maude [9]. Roughly speaking, a *rewriting logic theory* seamlessly combines a *term rewriting system* (TRS), together 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 [1, 13, 16, 17], but to the best of our knowledge, no practical animation tool for RWL/Maude has been formally developed. To debug Maude programs, Maude has a basic tracing facility that allows the user to advance through the program execution letting him/her select the statements to be traced, except for the application of algebraic axioms that are not

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

under user control and are never recorded as execution steps 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 below, 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 parametric (forward) exploration technique for RWL computations. 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*. The algorithm is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates RWL computations. Our formulation takes into account the precise way in which Maude mechanizes the rewriting process and revisits all those rewrite steps in an instrumented, 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 simply discarded.

Related Work. Program animators have existed since the early years of programming. Although several steppers have been implemented in the functional programming community (see [10] for references), none of these systems applies to the animation and forward slicing of Maude computations. An algebraic stepper for Scheme is defined and formally proved in [10], 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. It explains a program's execution as a sequence of reduction steps based on the ordinary laws of algebra for the functional core of the language and more general algebraic laws for the rest of the language. In order to discover all of the steps that occur during the program evaluation, the stepper rewrites (or "instruments") the code. The inserted code uses a mechanism called "continuation marks" to store information about the program's execution as it is running and makes calls to the stepper before, after, and during the evaluation of each program expression. Continuation marks allow the stepper to reuse the underlying Scheme implementation without having to re-implement the evaluator. The stepper's implementation technique also applies to both ML and Haskell since it supports states, continuations, multiple threads of control, and lazy evaluation [10].

In [4, 5], 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 discarded. 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 affect a 'selected computation". However, for the dual problem of "what program components might be affected by a selected computation", a kind of forward expansion is needed which has been overlooked to date.

Plan of the paper. Section 2 recalls some fundamental notions of RWL, and Section 3 summarizes the rewriting modulo equational theories defined in Maude. Section 4 formulates the exploration as a parameterized procedure that is completely controlled by the user, while Section 5 formalizes three different inspection techniques that are obtained as an instance of the generic scheme. Finally, Section 6 reports on a prototypical implementation of the proposed techniques, and Section 7 concludes.

2 Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [18] and Rewriting Logic [14]. Some familiarity with the Maude language [9] is also required.

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [9]. 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* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots\}$ 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:

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

For any substitution σ and set of variables V, $\sigma_{\upharpoonright V}$ denotes the substitution obtained from σ by restricting its domain to V (i.e., $\sigma_{\upharpoonright V}(x) = x\sigma$ if $x \in V$, otherwise $\sigma_{\upharpoonright V}(x) = x$). Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. 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* 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 is an expression of the form $[l] : \lambda \to \rho$, 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 $\lambda \to \rho$ (resp. equation $\lambda = \rho$).

3 Rewriting Modulo Equational Theories

An *order-sorted equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of (oriented) equations, and B is 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. The equational theory 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 set of rewrite rules.

Example 3.1

The following rewrite theory, encoded in Maude, specifies a close variant of the fault-tolerant client-server communication protocol of [15].

```
mod CLIENT-SERVER-TRANSF is inc INT + QID .
  sorts Content State Msg Cli Serv Addressee
    Sender Data CliName ServName Question Answer .
  subsort Msg Cli Serv < State .
  subsort Addressee Sender CliName ServName < Qid .
  subsort Question Answer Data < Nat .

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

```
  vars C S Addr : Qid .
  vars Q D A : Nat .
  var CNT : [Content] .

  eq [succ] : f(S, C, Q) = s(Q) .

  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]  : (Addr <- CNT) =>
               (Addr <- CNT) & (Addr <- CNT) .
  rl [loss]  : (Addr <- CNT) => null .
endm
```

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

The names of clients and servers belong to the sort `Qid`. Clients are represented as 4-tuples of the form `[C, S, Q, D]`, where `C` is the client's name, `S` is the name of the server it wants to communicate with, `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 been yet 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 `Addr <- CNT`, where `Addr` is the addressee's name, and `CNT` stands for the message contents. Such contents are pairs {`Addr`,`N`}, with `Addr` being the sender's name and `N` being a number that represents either a request or a response.

The server `S` uses a function `f` (only known to the server itself) that, given a question `Q` from client `C`, the call `f(S,C,Q)` computes the answer `s(Q)` where `s(Q)` is the successor of `Q`. This function is specified by means of the equation `succ`.

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. Indeed, the right-hand side [`C`, `S`, `Q`, `A`] of the rule 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 [`C`, `S`, `Q`, `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 [12] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [8]; that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is in general undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The exploration technique formalized in this work is formulated by considering the precise way in which Maude proves the rewrite steps (see Section 5.2 in [9]). 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. 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 normalized term $t{\downarrow_\Delta}$ to which no further equations can be applied. The term $t{\downarrow_\Delta}$ is called a *canonical form* of t w.r.t. Δ. On the other hand, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [9].

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, $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) 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 $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$).

Under appropriate conditions on the rewrite theory, a rewrite step modulo E on a term t can be implemented without loss of completeness by applying the following rewrite strategy [11]: (i) reduce t w.r.t. $\rightarrow_{\Delta,B}$ until the canonical form $t{\downarrow_\Delta}$ is reached; (ii) rewrite $t{\downarrow_\Delta}$ w.r.t. $\rightarrow_{R,B}$.

A *computation* $\mathcal{C}$ in the rewrite theory $(\Sigma, \Delta \cup B, R)$ is a rewrite sequence

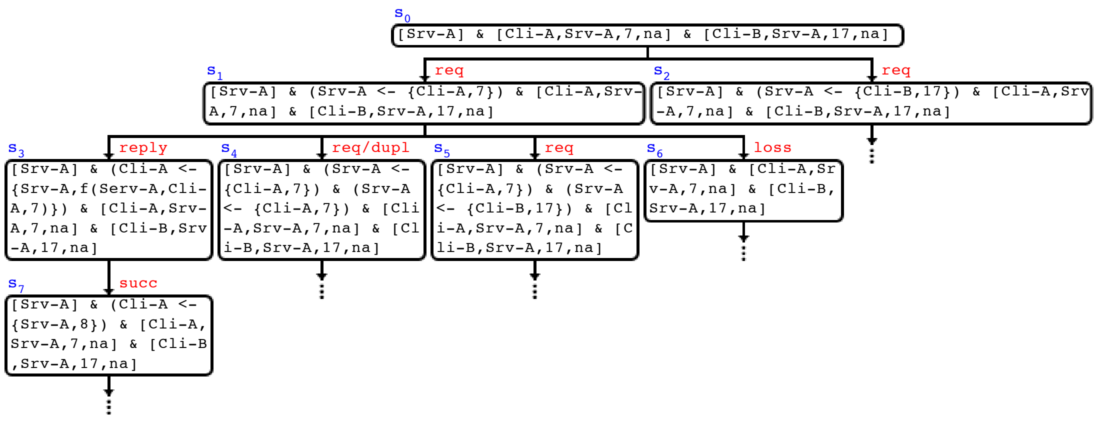
$$s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_\Delta} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_\Delta} \ldots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. Terms that appear in computations are also called (program) *states*.

A *computation tree* $\mathcal{T}_{\mathcal{R}}(s)$ for a term s and a rewrite theory $\mathcal{R}$ is a tree-like representation of all the possible computations in $\mathcal{R}$ that originate from the initial state s. More precisely, $\mathcal{T}_{\mathcal{R}}(s)$ is a labelled tree whose root node label identifies the initial state s and whose edges are labelled with rewrite steps of the form $t \rightarrow_{R\cup\Delta,B} t'$ so that any node in $\mathcal{T}_{\mathcal{R}}(s)$ represents a program state of a computation stemming from s.

Example 3.2

Consider the rewrite theory of Example 3.1 together with the initial state `[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 state and model interactions between clients `Cli-A` and `Cli-B` and server `Srv-A`. The computed tree is depicted in the following picture where, for the sake of simplicity, 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.


s0
[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
req
req
s1
[Srv-A] & (Srv-A <- {Cli-A,7}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
s2
[Srv-A] & (Srv-A <- {Cli-B,17}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
reply
req/dupl
req
loss
s3
[Srv-A] & (Cli-A <- {Srv-A,f(Serv-A,Cli-A,7)}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
s4
[Srv-A] & (Srv-A <- {Cli-A,7}) & (Srv-A <- {Cli-A,7}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
s5
[Srv-A] & (Srv-A <- {Cli-A,7}) & (Srv-A <- {Cli-B,17}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
s6
[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
succ
s7
[Srv-A] & (Cli-A <- {Srv-A,8}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]

Given a computation $\mathcal{C}$, it is always possible to expand $\mathcal{C}$ in an *instrumented* computation $\mathcal{C}_{inst}$ in which each application of the matching modulo B algorithm is mimicked by the explicit application of a suitable equational axiom, which is also oriented as a rewrite rule [3].

Also, typically hidden inside the B-matching algorithms, some flat/unflat transformations allow terms that contain operators that obey associative-commutative axioms to be rewritten. 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 [3]. For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b, f(f(b, a), c)) =_B f(f(b, c), f(a, b))$, we can represent it by using the "internal sequence" of transformations $f(b, f(f(b, a), c)) \overset{flat}{\longrightarrow}{}^*_B f(a, b, b, c) \overset{unflat}{\longrightarrow}{}^*_B f(f(b, c), f(a, b))$, where the first subsequence corresponds to a *flattening* transformation sequence that obtains the AC canonical form, while the second one corresponds to the inverse, unflattening transformation. This way, any given instrumented computation consists of a sequence of (standard) rewrites using the equations ($\to_\Delta$), rewrite rules ($\to_R$), equational axioms and flat/unflat transformations ($\to_B$). By abuse of notation, since equations and axioms are both interpreted as rewrite rules in our formulation, we often abuse the notation $\lambda \to \rho$ to denote rules as well as (oriented) equations and axioms. Given an instrumented computation $\mathcal{C}_{inst}$, by $|\mathcal{C}_{inst}|$ we denote its *length* —that is, the number of rewrite steps that occur in $\mathcal{C}_{inst}$. We use the functions $head(\mathcal{C}_{inst})$ and $tail(\mathcal{C}_{inst})$ to respectively select the first rewrite step in $\mathcal{C}_{inst}$ and the instrumented computation yielded by removing the first rewrite step from $\mathcal{C}_{inst}$. In the sequel, we also assume the existence of a function $instrument(\mathcal{C})$, which takes a computation $\mathcal{C}$ and delivers its instrumented counterpart.

Example 3.3

Consider the rewrite theory of Example 3.1 together with the following computation:

$$\begin{aligned}\mathcal{C} = \quad & \texttt{[Cli, Srv, 7, na] \& [Srv] \& Cli <- \{Srv, f(Srv, Cli, 7)\}} \to_{\Delta,B} \\ & \texttt{[Cli, Srv, 7, na]\& [Srv] \& Cli <- \{Srv, 8\}} \to_{R,B} \\ & \texttt{[Cli, Srv, 7 , 7] \& [Srv]}\end{aligned}$$

The corresponding instrumented computation $\mathcal{C}_{inst}$, which is produced by $instrument(\mathcal{C})$, is given by 1) suitably parenthesizing states by means of flattening/unflattening transformations when needed, and 2) making commutative "steps" explicit by using the (oriented) equational axiom (`X & Y` $\rightarrow$ `Y & X`) in B that models the commutativity property of the (juxtaposition) operator &. Note these transformations are needed to enable the application of the rule `rec` (of R) to the seventh state.

$$
\begin{array}{ll}
\mathcal{C}_{inst} = & \texttt{[Cli, Srv, 7, na] \& [Srv] \& Cli <- \{Srv, f(Srv, Cli, 7)\}} \xrightarrow{succ}_{\Delta} \\
& \texttt{[Cli, Srv, 7, na] \& [Srv] \& Cli <- \{Srv, 8\}} \xrightarrow{unflat}_{B} \\
& \texttt{[Cli, Srv, 7, na] \& ([Srv] \& Cli <- \{Srv, 8\})} \xrightarrow{comm}_{B} \\
& \texttt{[Cli, Srv, 7, na] \& (Cli <- \{Srv, 8\} \& [Srv])} \xrightarrow{flat}_{B} \\
& \texttt{[Cli, Srv, 7, na] \& Cli <- \{Srv, 8\} \& [Srv]} \xrightarrow{unflat}_{B} \\
& \texttt{([Cli, Srv, 7, na] \& Cli <- \{Srv, 8\}) \& [Srv]} \xrightarrow{comm}_{B} \\
& \texttt{(Cli <- \{Srv, 8\} \& [Cli, Srv, 7, na]) \& [Srv]} \xrightarrow{rec}_{R} \\
& \texttt{[Cli, Srv, 7, 7] \& [Srv]}
\end{array}
$$

4 Exploring the Computation Tree

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.

However, one may still be interested in analysing a fragment of a given computation tree for debugging or program comprehension purposes. This section presents an exploration technique that allows the user to incrementally generate and inspect a portion $\mathcal{T}^{\bullet}_{\mathcal{R}}(s^{\bullet})$ of a computation tree $\mathcal{T}_{\mathcal{R}}(s)$ by expanding (fragments of) its program states into their descendants starting from the root node. The exploration is an interactive procedure that is completely controlled by the user, who is free to choose the program state fragments to be expanded.

4.1 Expanding a Program State

A state *fragment* of a state s is a term $s^{\bullet}$ that hides part of the information in s, that is, the irrelevant data in s are simply replaced by special $\bullet$-variables of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$. Given a state fragment $s^{\bullet}$, 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 state fragment are called *meaningful* symbols. By $\mathcal{V}ar^{\bullet}(exp)$ we define the set of all $\bullet$-variables that occur in the expression *exp*.

Basically, a state fragment records just the information the user wants to observe of a given program state.

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

Definition 4.1 *Let $t \in \tau(\Sigma, \mathcal{V})$ be a state, and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. Then, the function fragment(t, P) is defined as follows.*

$$fragment(t, P) = recfrag(t, P, \Lambda), \ \ where$$

```
function $inspect(s^\bullet, \mathcal{C}_{instr}, \mathcal{I})$
1. if $|\mathcal{C}_{instr}| = 1$ then
2.   return $\mathcal{I}(s^\bullet, \mathcal{C}_{instr})$
3. elseif $|\mathcal{C}_{instr}| > 1$
4.   if $\mathcal{I}(s^\bullet, head(\mathcal{C}_{instr}))! =$ nil then
5.     return $inspect(\mathcal{I}(s^\bullet, head(\mathcal{C}_{instr})),$
                       $tail(\mathcal{C}_{instr}), \mathcal{I})$
6.   else
7.     return nil
8.   end
9. end
endf
```

Figure 1: The *inspect* function.

```
function $expand(s^\bullet, s, \mathcal{R}, \mathcal{I})$
1. $\mathcal{E}^\bullet_{\mathcal{R}} = \emptyset$
2. for each $s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t$
              with $w \in \mathcal{MP}os(s^\bullet)$
3.   $\mathcal{C}_{inst} = instrument(s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t)$
4.   $t^\bullet = inspect(s^\bullet, \mathcal{C}_{inst}, \mathcal{I})$
5.   if $t^\bullet \neq$ nil then $\mathcal{E}^\bullet_{\mathcal{R}} = \mathcal{E}^\bullet_{\mathcal{R}} \cup \{s^\bullet \overset{r}{\rightarrow} t^\bullet\}$
6. end
7. return $\mathcal{E}^\bullet_{\mathcal{R}}$
endf
```

Figure 2: The *expand* function.

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

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

Roughly speaking, *fragment*(t, P) yields a state fragment of t w.r.t. a set of positions P that includes all symbols of t that occur within the paths from the root to any position in P, while each maximal subterm $t_{|p}$, with $p \notin P$, is replaced by means of a freshly generated $\bullet$-variable.

Example 4.2

Let $t = d(f(g(a, h(b)), c), a)$ be a state, and let $P = \{1.1,\ 1.2\}$ be a set of positions of t. By applying Definition 4.1, we get the state fragment $t^\bullet = fragment(t, P) = d(f(g(\bullet_1, \bullet_2), c), \bullet_3)$ and the set of meaningful positions $\mathcal{MP}os(t^\bullet) = \{\Lambda, 1, 1.1, 1.2\}$.

An *inspection criterion* is a function $\mathcal{I}(s^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t)$ that, given a rewrite step $s \overset{r,\sigma,w}{\rightarrow}_K t$, with $K \in \{\Delta, R, B\}$ and a state fragment $s^\bullet$ of s, computes a state fragment $t^\bullet$ of the state t. Roughly speaking, an inspection criterion controls the information content conveyed by term fragments resulting from the execution of standard rewrite steps. Hence, distinct implementations of the inspection criteria $\mathcal{I}(s^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t)$ may produce distinct fragments $s^\bullet \overset{r}{\rightarrow} t^\bullet$ of the considered rewrite step. We assume that the special value **nil** is returned by the inspection criterion, whenever no fragment $t^\bullet$ can be delivered. Several examples of inspection criteria are shown in Section 5.

The function *inspect* of Figure 1 allows an inspection criterion $\mathcal{I}$ to be sequentially applied along an instrumented computation $\mathcal{C}_{instr}$ in order to generate the fragment of the last state of the computation. Specifically, given an instrumented computation $s_0 \rightarrow_K s_1 \rightarrow_K \ldots \rightarrow s_n$, $n > 0$, the computation is traversed and the inspection criterion $\mathcal{I}$ is recursively applied on each rewrite step $s_i \rightarrow_K s_{i+1}$ w.r.t. the input fragment $s^\bullet_i$ to generate the next fragment $s^\bullet_{i+1}$.

The expansion of a single program state is specified by the function *expand*$(s^\bullet, s, \mathcal{R}, \mathcal{I})$ of Figure 2, which takes as input a state s and its fragment $s^\bullet$ to be expanded w.r.t. a rewrite theory $\mathcal{R}$ and an inspection criterion $\mathcal{I}$. Basically, *expand* unfolds the state fragment $s^\bullet$ w.r.t. all the possible rewrite steps $s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t$ that occur at the meaningful positions of $s^\bullet$ and stores the corresponding fragments of $s^\bullet \overset{r}{\rightarrow} t^\bullet$ in the set $\mathcal{E}^\bullet$. Note that, to compute the state fragment $t^\bullet$ for a rewrite step $s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t$ and a state fragment $s^\bullet$, *expand* first generates the instrumented computation $\mathcal{C}_{inst}$ of the considered step and then applies the inspection criterion $\mathcal{I}$ over $\mathcal{C}$ by using the *inspect* function.

4.2 Computing a Fragment of the Computation Tree

The construction of a fragment $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$ of a computation tree $\mathcal{T}_\mathcal{R}(s_0)$ is specified by the function *explore* given in Figure 3. Essentially, *explore* formalizes an interactive procedure that starts from a tree fragment (built using the auxiliary function *createTree*), which only consists of the root node $s^\bullet_0$, and repeatedly uses the function *expand* to compute rewrite step fragments that correspond to the visited tree edges, w.r.t. a given inspection criterion. The tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$ is built by choosing, at each loop iteration of the algorithm, the tree node that represents the state fragment to be expanded by means of the auxiliary function *pickLeaf*$(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0))$, which allows the user to freely select a node $s^\bullet$ from the frontier of the current tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$. Then, $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$ is augmented by calling *addChildren*$(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0), s^\bullet, \mathit{expand}(s^\bullet, s, \mathcal{R}, \mathcal{I}))$. This function call adds all the edges $s^\bullet \to t^\bullet$, which are obtained by expanding $s^\bullet$, to the tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$.

The special value **EoE** (End of Exploration) is used to terminate the inspection process: when the function *pickLeaf*$(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0))$ is equal to **EoE**, no state to be expanded is selected and the exploration terminates delivering the computed fragment $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$.

```
function $\mathit{explore}(s^\bullet_0, s_0, \mathcal{R}, \mathcal{I})$
1. $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0) = \mathit{createTree}(s^\bullet_0)$
2. while($s^\bullet = \mathit{pickLeaf}(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)) \neq$ EoE) do
3.   $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0) = \mathit{addChildren}(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0), s^\bullet, \mathit{expand}(s^\bullet, s, \mathcal{R}, \mathcal{I}))$
4. od
5. return $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$
endf
```

Figure 3: The *explore* function.

5 Particularizing the Exploration

The methodology given in Section 4 provides a generic scheme for the exploration of computation trees w.r.t. a given inspection criterion $\mathcal{I}$ that must be 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 computations of interest can be animated. In the second case, we implement a partial stepper that allows computations with partial inputs to be animated. Finally, in the last example, the chosen inspection criterion implements an automated, forward slicing technique that allows relevant control and data information to be extracted from computation trees.

5.1 Interactive Stepper

Given a computation tree $\mathcal{T}_\mathcal{R}(s_0)$ for an initial state 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 4 with the inspection criterion

$$\mathcal{I}_{step}(s^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t) = t$$

that always returns the reduced state t of the rewrite step $s \overset{r,\sigma,w}{\to}_K t$, with $K \in \{\Delta, R, B\}$.

This way, by starting the exploration from a state fragment that corresponds to the whole initial state s_0 (i.e., $s_0^\bullet = s_0$), the call $\mathit{explore}(s_0, \mathcal{R}, \mathcal{I}_{step})$ generates a fragment $\mathcal{T}_{\mathcal{R}}^\bullet(s_0^\bullet)$ of the 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 3.1 and the computation tree in Example 3.2. Assume the user starts the exploration by calling $\mathit{explore}(s_0, \mathcal{R}, \mathcal{I}_{step})$, which allows the first level of the computation tree to be unfolded by expanding the initial state s_0 w.r.t. the inspection criterion $\mathcal{I}_{step}$. This generates the tree $\mathcal{T}_{\mathcal{R}}^\bullet(s_0)$ containing the edges $\{s_0 \overset{req}{\to} s_1, s_0 \overset{req}{\to} s_2\}$. Now, if the user carries on with the exploration of program state s_1 and then quits, s_1 will be expanded and the tree $\mathcal{T}_{\mathcal{R}}^\bullet(s_0)$ will be augmented accordingly. Specifically, the resulting $\mathcal{T}_{\mathcal{R}}^\bullet(s_0)$ will include the following edges $\{s_0 \overset{req}{\to} s_1, s_0 \overset{req}{\to} s_2, s_1 \overset{succ}{\to} s_3, s_1 \overset{req}{\to} s_4, s_1 \overset{dupl}{\to} s_4, s_1 \overset{req}{\to} s_5, s_1 \overset{loss}{\to} s_6\}$.

It is worth noting that all the program state fragments produced by the program stepper defined above are "concrete" (i.e. state fragments that do not include •-variables). However, sometimes it may be useful to work with partial information and hence with state fragments that abstract "concrete" program states by using •-variables. This approach may help to focus user's attention on the parts of the program states that the user wants to observe, disregarding unwanted information and useless rewrite steps.

Example 5.2

Consider the following two rewrite rules $[r_1] : f(x, b) \to g(x)$ and $[r_2] : f(a, y) \to h(y)$ together with the initial state $f(a, b)$. Then, the computation tree in this case is finite and only contains the tree edges $f(a, b) \overset{r_1,\{x/a\},\Lambda}{\to} g(a)$ and $f(a, b) \overset{r_2,\{y/b\},\Lambda}{\to} h(b)$. Now, consider the state fragment $f(\bullet_1, b)$, where only the second input argument is relevant. If we decided to expand the initial state fragment $f(\bullet_1, b)$, we would get the tree fragment represented by the single rewrite step $f(\bullet_1, b) \overset{r_1}{\to} g(\bullet_1)$, since the partial input encoded into $f(\bullet_1, b)$ cannot be rewritten via r_2.

In light of Example 5.2 and our previous considerations, we define the following inspection criterion

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

Roughly speaking, given a rewrite step $\mu : s \overset{r,\sigma,w}{\to}_K t$, with $K \in \{\Delta, R, B\}$, the criterion $\mathcal{I}_{pstep}$ returns a state fragment $t^\bullet$ of the reduced state t, whenever $s^\bullet$ can be rewritten to $t^\bullet$ using the very same rule r and position w that occur in μ.

The particularization of the exploration scheme with 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 fragments of the computation trees.

5.2 Forward Trace Slicer

Forward trace slicing is a program analysis technique that allows computations to be simplified w.r.t. a selected fragment of their initial state. More precisely, given a computation $\mathcal{C}$ with initial state s_0 and a state fragment $s_0^\bullet$ of s_0, forward slicing yields a simplified view $\mathcal{C}^\bullet$ of $\mathcal{C}$ in which each state s of the original computation is replaced by a state fragment $s^\bullet$ that only

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

Figure 4: Inspection criterion that models forward slicing of a rewrite step

records the information that depends on the meaningful symbols of $s_0^\bullet$, while unrelated data are simply pruned away.

In the following, we define an inspection criterion $\mathcal{I}_{slice}$ that implements the forward slicing of a single rewrite step. The considered criterion takes two parameters as input, namely, a rewrite step $\mu = (s \overset{r,\sigma,w}{\to}_K t)$ (with $r = \lambda \to \rho$ and $K \in \{\Delta, R, B\}$) and a state fragment $s^\bullet$ of a state s. It delivers the state fragment $t^\bullet$ which includes only those data that are related to the meaningful symbols of $s^\bullet$. Intuitively, the state fragment $t^\bullet$ is obtained from $s^\bullet$ by "rewriting" $s^\bullet$ at position w with the rule r and a suitable substitution that abstracts unwanted information of the computed substitution with $\bullet$-variables. A rigorous formalization of the inspection criterion $\mathcal{I}_{slice}$ is provided by the algorithm in Figure 4.

Note that, by adopting the inspection criterion $\mathcal{I}_{slice}$, the exploration scheme of Section 4 automatically turns into an interactive, forward trace slicer that expands program states using the slicing methodology encoded into the inspection criterion $\mathcal{I}_{slice}$. In other words, given a computation tree $\mathcal{T}_\mathcal{R}(s_0)$ and a user-defined state fragment $s_0^\bullet$ of the initial state s_0, any branch $s_0^\bullet \to s_1^\bullet \ldots \to s_n^\bullet$ in the tree $\mathcal{T}^\bullet_\mathcal{R}(s_0^\bullet)$, which is computed by the *explore* function, is the sliced counterpart of a computation $s_0 \to_{R\cup\Delta,B} s_1 \ldots \to_{R\cup\Delta,B} s_n$ (w.r.t. the state fragment $s_0^\bullet$) that appears in the 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 $s^\bullet$ (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 state fragment $s^\bullet$ and, hence, the function returns the **nil** 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.

Definition 5.3 (substitution update) *Let σ_1 and σ_2 be two substitutions. The* update *of σ_1 w.r.t. σ_2 is defined by the operator $\langle\!|\_,\_|\!\rangle$ as follows:*
$\langle\!|\sigma_1, \sigma_2|\!\rangle = \sigma_{\restriction Dom(\sigma_1)}$, *where*

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

The operator $\langle\!|\sigma_1, \sigma_2|\!\rangle$ updates (overrides) a substitution σ_1 with a substitution σ_2, where both σ_1 and σ_2 may contain $\bullet$-variables. The main idea behind $\langle\!|\_,\_|\!\rangle$ is that, for the slicing of the rewrite step μ, all variables in the applied rewrite rule r are naïvely assumed to be initially bound to irrelevant data $\bullet$, and the bindings are incrementally updated as we apply the rule r.

More specifically, we initially define the substitution $\theta = \{x/fresh^{\bullet} \mid x \in Var(\rho)\}$ that binds each variable in $\lambda \to \rho$ 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.

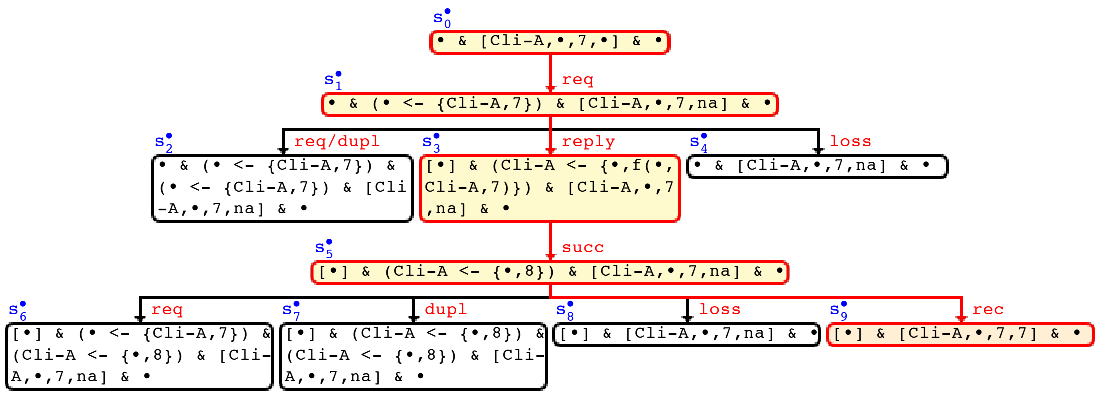
We first compute the state fragment $\lambda^{\bullet} = fragment(\lambda, \mathcal{MP}os(\mathcal{V}ar^{\bullet}(s^{\bullet}_{|w})) \cap \mathcal{P}os(\lambda))$ that only records the meaningful symbols of the left-hand side λ of the rule r w.r.t. the set of meaningful positions of $s^{\bullet}_{|w}$. Then, by matching $\lambda^{\bullet}$ with $s^{\bullet}_{|w}$, 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 tracked. Finally, the state fragment $t^{\bullet}$ 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 fragment of the resulting state $t^{\bullet}$.

Example 5.4

Consider the computation tree of Example 3.2 whose initial state is

$$s_0 = \texttt{[Srv-A] \& [Cli-A,Srv-A,7,na] \& [Cli-B,Srv-A,17,na]}.$$

Let $s^{\bullet}_0 = \bullet$ `&` `[Cli-A,`$\bullet$`,7,` $\bullet$`]` `&` $\bullet$ be a state fragment[1] of s_0 where only request `7` of Client `Cli-A` is considered of interest. By sequentially expanding the nodes $s^{\bullet}_0$, $s^{\bullet}_1$, $s^{\bullet}_3$, and $s^{\bullet}_5$ w.r.t. the inspection criterion $\mathcal{I}_{slice}$, we get the following fragment of the given computation tree:



s0•
• & [Cli-A,•,7,•] & •
req
s1•
• & (• <- {Cli-A,7}) & [Cli-A,•,7,na] & •
req/dupl
s2•
• & (• <- {Cli-A,7}) & (• <- {Cli-A,7}) & [Cli-A,•,7,na] & •
reply
s3•
[•] & (Cli-A <- {•,f(•,Cli-A,7)}) & [Cli-A,•,7,na] & •
loss
s4•
• & [Cli-A,•,7,na] & •
succ
s5•
[•] & (Cli-A <- {•,8}) & [Cli-A,•,7,na] & •
req
s6•
[•] & (• <- {Cli-A,7}) & (Cli-A <- {•,8}) & [Cli-A,•,7,na] & •
dupl
s7•
[•] & (Cli-A <- {•,8}) & (Cli-A <- {•,8}) & [Cli-A,•,7,na] & •
loss
s8•
[•] & [Cli-A,•,7,na] & •
rec
s9•
[•] & [Cli-A,•,7,7] & •

Note that the slicing process automatically computes a tree fragment that represents a partial view of the protocol interactions from client `Cli-A`'s perspective. Actually, irrelevant information is hidden and rules applied on irrelevant positions are directly ignored, which allows a simplified fragment to be obtained favoring its inspection for debugging and analysis purposes. In fact, if we observe the highlighted computation in the tree, we can easily detect the wrong behaviour of the rule `rec`. Specifically, by inspecting the state fragment $s^{\bullet}_9 = ([\bullet]$ `&` $[\texttt{Cli}-\texttt{A}, \bullet, 7, 7]$ `&` $\bullet)$, which is generated by an application of the rule `rec`, we immediately realize that the response `8` produced in the parent state $s^{\bullet}_5$ has not been stored in $s^{\bullet}_9$, which clearly indicates a buggy implementation of the considered rule.

[1]Throughout the example, we omit indices from the considered $\bullet$-variables to keep notation lighter and improve readability. So, any variable $\bullet_i$, $i = 0, 1, \ldots$, is simply denoted by $\bullet$.

Finally, it is worth noting that the forward trace slicer implemented via the criterion $\mathcal{I}_{slice}$ differs from the partial stepper given at the end of Section 5.1. Given a state fragment $s^{\bullet}$ and a rewrite step $\overset{r,\sigma,w}{\to}_K t$, $\mathcal{I}_{slice}$ always yields a fragment $t^{\bullet}$ when the rewrite occurs at a meaningful position. By contrast, the inspection criterion $\mathcal{I}_{pstep}$ encoded in the partial stepper may fail to provide a computed fragment $t^{\bullet}$ when $s^{\bullet}$ does not rewrite to $t^{\bullet}$.

Example 5.5

Consider the same rewrite rules and initial state $f(\bullet_1, b)$ of Example 5.2. By expanding $f(\bullet_1, b)$ w.r.t. the inspection criterion $\mathcal{I}_{slice}$, we get the computation tree fragment with tree edges $f(\bullet_1, b) \overset{r_1}{\to} g(\bullet_1)$ and $f(\bullet_1, b) \overset{r_2}{\to} h(b)$, whereas the partial stepper only computes the tree edge $f(\bullet_1, b) \overset{r_1}{\to} g(\bullet_1)$ as shown in Example 5.2.

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 150 Maude function definitions (approximately 1600 lines of source code).

Anima also comes with an intuitive Web user interface based on AJAX technology, which allows users to graphically display and animate computation tree fragments. 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 5 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 and JSP. 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 Database component.

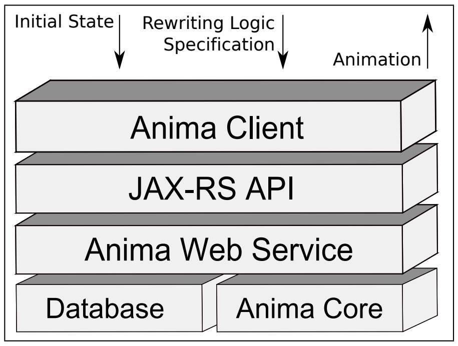


Figure 5: Anima architecture.

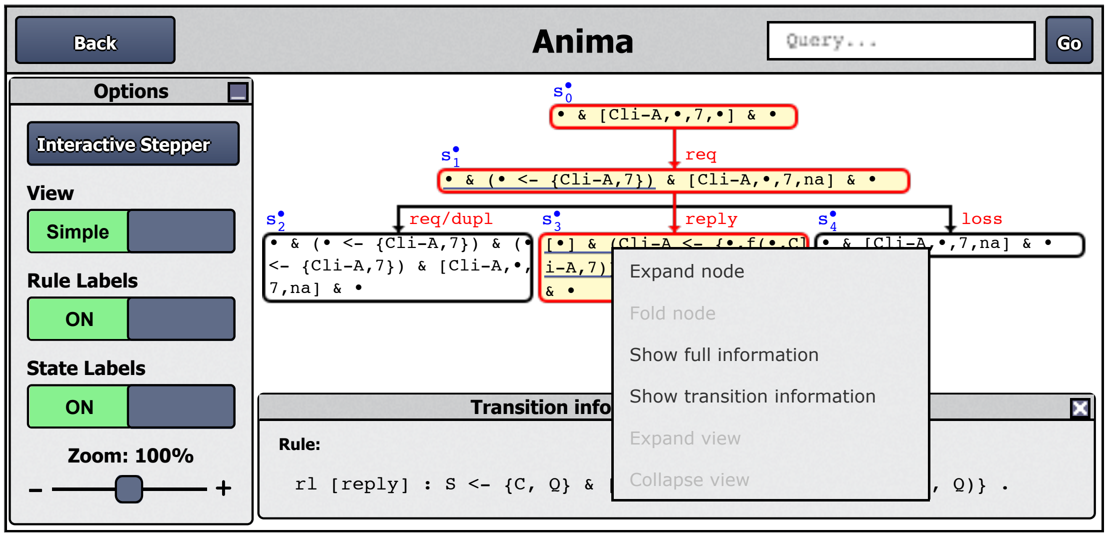
A screenshot that shows the Anima tool at work on the case study that is described in Example 5.4 is given in Figure 6.

These are the main features provided by Anima:

1. *Inspection strategies.* The tool implements the three inspection strategies described in Section 5. As shown in Figure 6, the user can select a strategy by using the selector provided in the option pane.

2. *Expanding/Folding program states.* The user can expand or fold states by right-clicking on them with the mouse and by selecting the *Expand/Fold Node* options from the contextual menu. For instance, in Figure 6, a state fragment on the frontier of the computed tree has been selected and is ready to be expanded through the *Expand Node* option.

3. *Selecting meaningful symbols.* State fragments can be specified by highlighting the state symbols of interest either directly on the tree or in the detailed information window.



```
• & [Cli-A,•,7,•] & •
• & (• <- {Cli-A,7}) & [Cli-A,•,7,na] & •
• & (• <- {Cli-A,7}) & (•
<- {Cli-A,7}) & [Cli-A,•,
7,na] & •
[•] & (Cli-A <- {•,f(•,C
i-A,7)
& •
• & [Cli-A,•,7,na] & •
rl [reply] : S <- {C, Q} & [ , Q)} .
```

Figure 6: Anima at work.

4. *Search mechanism.* The search facility implements a pattern language that allows state information of interest to be searched on huge states and 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.

5. *Transition information.* Anima facilitates the inspection of a selected rewrite step $s \to t$ that occurs in the computation tree by underlining its redex in s and the reduced subterm in t. Some additional transition information is also displayed in the *transition information window* (e.g., the rule/equation applied, the rewrite position, and the computed substitution of the considered rewrite step) by right-clicking on the corresponding option.

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 concretely demonstrates 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.

References

[1] M. Alpuente, D. Ballis, M. Baggi, and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In *Proc. 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*, pages 43–52. ACM, 2010.

[2] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Model-checking Web Applications with Web-TLR. In *Proc. of 8th Int'l Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *LNCS*, pages 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*, volume 6803 of *LNCS/LNAI*, pages 34–48. Springer-Verlag, 2011.

[4] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Backward Trace Slicing for Conditional Rewrite Theories. In *Proc. LPAR-18*, volume 7180 of *LNCS*, pages 62–76. Springer-Verlag, 2012.

[5] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Slicing-Based Trace Analysis of Rewriting Logic Specifications with iJulienne. In Matthias Felleisen and Philippa Gardner, editors, *Proc. of 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 121–124. Springer, 2013.

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

[7] M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In *Proc. of 7th Int'l Conference on Computational Methods in Systems Biology (CMSB '09)*, volume 5688 of *LNCS*, pages 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. 10th European Symposium on Programming*, volume 2028 of *LNCS*, pages 320–334. Springer-Verlag, 2001.

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

[12] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

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

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

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

[16] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative Debugging of Rewriting Logic Specifications. In *Recent Trends in Algebraic Development Techniques, 19th Int'l Workshop, WADT 2008*, volume 5486 of *LNCS*, pages 308–325. Springer-Verlag, 2009.

[17] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative Debugging of Missing Answers for Maude. In *21st Int'l Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *LIPIcs*, pages 277–294. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[18] TeReSe, editor. *Term Rewriting Systems.* Cambridge University Press, Cambridge, UK, 2003.