UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Slicing Slices

An Incremental Backward Trace Slicing Methodology for RWL Computations

Master's Thesis

Presented by: Julia Sapiña Sanchis

Supervisors: María Alpuente Frasnedo
Francisco Frechina Navarro

Valencia – July 10th, 2013.

Abstract

Execution traces are an important source of information for program understanding and debugging. However, they have as an important drawback that they are commonly huge and complex, making their manual analysis unfeasible.

In this master's thesis, we develop a trace slicing technique for rewriting logic computations together with its corresponding implementation in the *i*Julienne graphical tool. Such technique gives support to the analysis of complex, textually-large system computations in rewriting logic, which is a general framework efficiently implemented in the Maude language. Given a Maude execution trace $\mathcal{T}$ and a slicing criterion for the trace (i.e., a piece of information that we want to observe in the final computation state), we traverse $\mathcal{T}$ from back to front and the backward dependence of the observed information is incrementally computed at each execution step. At the end of the traversal, a simplified trace slice is obtained by filtering out all the irrelevant data that do not impact on the data of interest. By simplifying the trace, the *i*Julienne tool, which implements the proposed trace slicing technique, favors better inspection and debugging activities. *i*Julienne is also endowed with a trace querying mechanism that increases flexibility and reduction power and allows program runs to be examined at the appropriate level of abstraction.

Keywords: *trace slicing, rewriting logic, debugging, Maude.*

Resumen

Las trazas de ejecución son una importante fuente de información para el entendimiento y depuración de los programas. Sin embargo, poseen el importante inconveniente de ser habitualmente enormes y complejas, haciendo inviable su análisis manual.

En esta tesis de máster se ha desarrollado una técnica de fragmentación de trazas para computaciones en lógica de reescritura junto con su correspondiente implementación en la herramienta gráfica *i*Julienne. Dicha técnica da soporte al análisis de extensas y complejas computaciones en lógica de reescritura, que es un marco formal muy general eficientemente implementado en el lenguaje Maude. Dada una traza de ejecución de Maude $\mathcal{T}$ y un criterio de fragmentación para la traza (es decir, un fragmento de información que se desea observar en el estado final de la computación), nuestra técnica recorre $\mathcal{T}$ de atrás hacia adelante a la vez que calcula la dependencia hacia atrás de la información observada de forma incremental en cada paso de ejecución. Al final del recorrido se obtiene una traza fragmentada simplificada al filtrar toda la información irrelevante que no afecta a los datos de interés. Al simplificar la traza, la herramienta *i*Julienne, que implementa la técnica de fragmentado propuesta, favorece un mejor análisis y depuración de las trazas manipuladas. *i*Julienne está además dotada de un mecanismo de consulta de la traza que incrementa la flexibilidad y poder de reducción y permite examinar las ejecuciones de programas en el nivel apropiado de abstracción.

Palabras clave: *fragmentación de trazas, lógica de reescritura, depuración, Maude.*

Resum

Les traces d'execució són una important font d'informació per a l'enteniment i depuració dels programes. No obstant això, posseeixen l'important inconvenient de ser habitualment enormes i complexes, fent inviable la seua anàlisi manual.

En aquesta tesi de màster s'ha desenvolupat una tècnica de fragmentació de traces per computacions en lògica de reescriptura juntament amb la seua corresponent implementació en l'eina gràfica *i*Julienne. L'esmentada tècnica dóna suport a l'anàlisi de complexes i extenses computacions en lògica de reescriptura, que és un marc formal molt general eficientment implementat en el llenguatge Maude. Donada una traça d'execució de Maude $\mathcal{T}$ i un criteri de fragmentació per la traça (és a dir, un fragment d'informació que es vol observar en l'estat final de la computació), la nostra tècnica es recorre $\mathcal{T}$ de darrere cap endavant al mateix temps que ses calcula la dependència cap enrere de la informació observada es calcula de forma incremental en cada pas d'execució. Al final del recorregut s'obté una traça fragmentada simplificada en filtrar tota la informació irrellevant que no afecta les dades d'interès. En simplificar la traça, l'eina *i*Julienne, que implementa la tècnica de fragmentació proposta, afavoreix una millor anàlisi i depuració de les traces manipulades. L'eina *i*Julienne està a més dotada d'un mecanisme de consulta de la traça que incrementa la flexibilitat i poder de reducció i permet examinar les execucions de programes en el nivell apropiat d'abstracció.

Paraules clau: *fragmentació de traces, lògica de reescriptura, depuració, Maude.*

Contents

List of Figures

Introduction

As professor Sir Maurice Wilkes, winner of the 2nd ACM A.M. Turing Award in 1967, recalled in his memoirs [Wil85], it is much more difficult to avoid making mistakes when programming than might be expected at first. Usually, programs do not work correctly the first time they are run and they must be revised in search for errors. It is this process of revising a program that gave birth to debugging.

A variety of debugging techniques have been developed ever since the pioneers of software development first realized of this tendency to make mistakes when programming. However, in most of these techniques there is usually a lot more information than the strictly needed to identify and reproduce the source of error. One may think that the more information we could get about a problem, the better, yet when the amount of information exceeds a certain limit it becomes a major drawback by obscuring the concrete information that cause the error. The consequences of this excess of information may vary from simple annoyance to the far more troubling inability to perform the required revision.

To help relieve this problem, Mark Weiser originally introduced in [Wei79, Wei81, Wei82] the concept of program slicing, which resulted in the development of a great variety of techniques based on it thenceforth. Given a slicing criterion (i.e., a piece of information initially considered relevant), slicing generally focus on identify all of its dependencies to disclose the most relevant information that may went unnoticed, discarding all the other irrelevant information in the process. Philosophy underlying slicing techniques can be

applied to different kinds of information (e.g., programs, execution traces, counter-examples offered by some model-checkers) that can be explored in many ways (e.g., forward, backward), leading to the different variants of slicing known nowadays.

The analysis of execution traces plays an important role in many program analysis approaches. Software systems commonly generate large and complex execution traces, whose analysis (or even simple inspection) is extremely time-consuming, and in some cases unfeasible to perform by hand. Standard tracers usually present execution histories that mainly consist of low-level execution steps so that the relationship between the executed program and the execution history is not easy to derive because some key dependencies that are naturally expressed at the programming language level can be either scattered or omitted in the trace. This is particularly true for those systems that are specified in Rewriting Logic (RWL) [Mes90b, Mes90a, Mes91], a very general *logical* and *semantic framework* that is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [BBF09] and Web systems [ABER10, ABR09]). Trace slicing [ABER11, ABFR12a] is a technique for reducing the size of execution traces by focusing on selected execution aspects, which makes it suitable for trace analysis and monitoring [CR09].

Rewriting logic is efficiently implemented in the high–performance language Maude [CDE$^+$02, CDE$^+$11]. In Maude, the states of a system are modeled as algebraic entities by using conditional equations, whereas the system's behavior is formalized via conditional rewrite rules that describe state transitions. These transitions are performed *modulo* equational theories that may also contain algebraic axioms such as commutativity, associativity and unity. The fact that RWL theories include both equations and rewrite rules provides a useful abstraction dial to find the right balance between abstraction and computational observability.

Execution traces generated by Maude are complex objects to deal with. Traces typically include thousands of rewrite steps that are obtained by applying the equations and rules of the considered specification (including all the internal rewrite steps for evaluating the conditions of such equations/rules). In addition, Maude traces are incomplete because algebraic axiom applications, which implicitly occur in an equational simplification process that is hidden within Maude's *matching modulo* algorithm, are not recorded at all in the trace. This provides a very low-level blueprint of program execution whose manual inspection is frequently unfeasible or, in the best case, is

an extremely labor-intensive and time-consuming task. Eventually, this implies that when an erroneous intermediate result in the trace is discovered, it is difficult to determine where the incorrect inference started.

To debug Maude programs, Maude has a tracing facility that allows the execution sequence to be traced, and is very customizable: it provides some control over conditions and allows the user to select the statements being applied at each step. A main difference with the trace slicing methodology described in [ABFR12a, ABFS13b] is that the tracer of Maude allows the trace size to be reduced by manually focusing on statements, while slicing is automatic and focuses on terms. Moreover, since each small rewrite step that is obtained by applying a single conditional equation, equational axiom or rule is shown in the trace, the user can easily miss the general view, and when the user detects an incorrect intermediate result, it is difficult to know where the incorrect inference started.

The formulation in [ABFR12a] takes into account the precise way in which Maude mechanizes the conditional rewriting process and revisits all those rewrite steps backwards in an instrumented, fine-grained way where each small step corresponds to the application of an equation (conditional equation or equational axiom) or rule. This allows to slice the input execution trace with regard to the set of symbols of interest (target symbols) by tracing back the target symbols along the execution trace so that all data that are not antecedents of the observed symbols are simply discarded. In this regard, the trace slices computed by the technique can be very helpful in debugging, since they only consist of the information that is strictly needed to deliver a critical part of the result (see discussion in [ABE+11]).

Slicing techniques as varied as they are, have in common the starting point given by the slicing criterion. Nevertheless, the slicing criterion may not be clear at the beginning of the slicing process and may need to be refined in order to discard the maximum irrelevant information possible with respect to the sought error. To cope with such cases, we have refined in [ABFS13b] our methodology based on [ABFR12a] by adding incremental capabilities to the slicing process, allowing the user to refine the slicing criterion at each step of the trace slicing process or even change it completely at any point.

Related Work

Tracing techniques have been extensively used in the past in functional debugging [CRW00]. Hat [CRW00] is an interactive debugging system that enables a computation to be explored backwards, starting from the program output or an error message (with which the computation aborted). Backward tracing in Hat is carried out by navigating a redex trail (i.e., a graph-like data structure that records dependencies among function calls), whereas the tracing technique described in this thesis does not require handling any supplementary data structure.

In recent years, the debugging and optimization techniques based on RWL have received growing attention. There exist very few approaches that address the problem of tracing rewrite sequences in term rewrite systems [ABER11, BKdV00, FT94, BKdV03], and all of them apply to unconditional systems. The techniques in [ABER11, BKdV00, BKdV03] rely on a labeling relation on symbols that allows data content to be traced back within the computation; this is achieved in [FT94] by formalizing a notion of dynamic dependence among symbols by means of contexts. In [BKdV00, BKdV03], non-left linear and collapsing rules are not considered or are dealt using ad-hoc strategies, while our approach requires no special treatment of such rules. Furthermore, only [ABER11] describes a tracing methodology for rewrite theories with rules, equations, sorts, and algebraic axioms. However, the only trace slicing technique that gives support to the analysis of RWL computations is [ABER11]. Given an execution trace $\mathcal{T}$, [ABER11] generates a trace slice of $\mathcal{T}$ w.r.t. a set of symbols of interest (target symbols) that appear in a given state of $\mathcal{T}$.

Unfortunately, the technique in [ABER11] is only applicable to *unconditional* RWL theories, and hence it cannot be employed when the source program includes conditional equations and/or rules since it would deliver incorrect and/or incomplete trace slices. The following example illustrates why conditions cannot be disregarded by the slicing process, which is what motivated [ABFR12a, ABFR12b] and gave rise to our last work [ABFS13b].

Example A

Consider the Maude specification of the function `_mod_` in Figure 1, which computes the reminder of the division of two natural numbers, and the associated execution trace `4 mod 5` $\rightarrow$ `4`. Assume that we are interested in observing the origins of the target symbol `4` that appears in the final state.

If we disregard the condition `Y > X` of the first conditional equation, the slicing technique of [ABER11] computes the trace slice `4 mod •` $\rightarrow$ `4`, which is not correct since there exists concrete instances of `4 mod •` that cannot be rewritten to `4` using the first rule (e.g., `4 mod 3` $\nrightarrow$ `4`).

```
mod M is inc NAT .
 var X  : Nat .
 var Y : NzNat .
 op _mod_ : Nat NzNat -> Nat .
 ceq X mod Y = X if Y > X .
 ceq X mod Y = (X - Y) mod Y
        if Y <= X .
endm
```

Figure 1: The `_mod_` operator

By contrast, the methodology of [ABFR12a, ABFS13b] produces the correct trace slice `4 mod 5` $\rightarrow$ `4`, since both arguments of `mod` are required to prove the rewrite step that introduces the symbol `4` in the final state.

Regarding slicing tools, among the first developed [HKF95] was *Spyder* [ADS93], which combined several techniques in order to assist the user in the debugging task. The slicing part of *Spyder* [Agr91], which was only capable to compute backward slices, was based on graph reachability and dependence graphs algorithms applied on the instrumented version of the original code being debugged [HKF95].

More recently, a source-level tracer for the functional programming language Haskell [HHJW07], has been developed in the *Hat* system [CRW00],. *Hat* grants the user access to exhaustive information about the observed computation by first executing the program to be debugged and storing the resulting trace in a file. Then, several slicing-based viewing tools (e.g., *Hat-trail*, *Hat-explore*, *Hat-detect*) can be used to explore the stored trace. Similarly to *Spyder*, this task is carried out by navigating a graph-like, supplementary data structure.

HaSlicer [RB06] is a prototype of a slicer for functional programs written also in Haskell that is used for the identification of possible coherent components from (functional) monolithic legacy code. Both backward and forward dependency slicing are covered by *HaSlicer*, which is proposed as a support tool for the software architect to manually improve program understanding,

and automatically discover software components. The latter is particularly useful as an architecture understanding technique in earlier phases of a re-engineering process.

Mercury [HCS+96] is a functional logic language that extends Haskell with logical capabilities and has his own slicing-based debugging tool, the *Mercury declarative debugger* [Mac05]. This debugger records an *annotated trace* that serves as source for constructing an EDT (evaluation dependence tree) that is searched by the main slicing algorithm implemented in the tool.

To the best of our knowledge, Julienne [ABFR12b] was the first trace slicing tool for Maude, which has been greatly improved in the *i*Julienne [ABFS13b] system, the new and improved implementation with incremental slicing capabilities presented in this thesis. In contrast to *Spyder* and *HaSlicer*, *i*Julienne is based on trace slicing rather than program slicing, and needs much less storage to perform flow-back analysis, as it requires neither the construction of data and control dependency graphs nor the creation and maintenance of the execution history.

Contributions

This master's thesis developes the first conditional trace slicing technique for RWL computations [ABFR12a] together with an enhanced trace slicing methodology called incremental slicing and its corresponding implementation [ABFS13b]. The proposed methodology is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates conditional RWL computations such as those delivered as counterexample traces by the Maude model-checker [BM12].

The contributions of this master's thesis can be summarized as follows:

- We have developed a novel and enhanced incremental trace slicing technique that we implemented in our tool *i*Julienne [ABFS13b].
- We present *i*Julienne, which is the first a slicing-based trace analysis tool that assists the user in the comprehension and debugging of RWL theories that are encoded in Maude. *i*Julienne is built on top of a trace slicer that implements the backward conditional trace slicing algorithm described in [ABER11, ABFR12a]. Roughly speaking, the trace slicing mechanism included in *i*Julienne rolls back the program execution (making *all* the rewrite and equational simplification steps

explicit) while tracking back only and all data in the trace that are needed to accomplish the selected slicing criterion, that is, the data that contribute to producing the set of *target symbols* that occur in the observed state of the trace. In other words, the trace slicer takes as input a Maude execution trace $\mathcal{T}$ and a slicing criterion $\mathcal{S}$ and yields as output a trace slice $\mathcal{T}^\star$ in which the pieces of information that are not antecedents of the selected target symbols in $\mathcal{S}$ are simply discarded from $\mathcal{T}$.

The core trace slicer included within *i*Julienne is a totally redesigned implementation of the slicing technique in [ABER11, ABFR12a] that supersedes and greatly improves the preliminary system presented in [ABFR12b]. The original algorithm implemented in [ABFR12b] was developed under the assumption that the user examines and slices the entire trace w.r.t. the fixed slicing criterion one time, in a non-incremental way. In contrast, the slicing criterion in *i*Julienne can be dynamically changed, which allows the user a step-wise focus on the information that the user wants to observe at any rewrite step.

The main features provided by the trace analyzer *i*Julienne are listed below.

(a) *i*Julienne is equipped with an *incremental* backward trace slicing algorithm that supports step-wise refinements of the trace slice and achieves huge reductions in the size of the trace. Starting from a Maude execution trace $\mathcal{T}$, a slicing criterion $\mathcal{S}$ can be attached to any given state of the trace and the computed trace slice $\mathcal{T}^\star$ can be repeatedly refined by applying backward trace slicing w.r.t. increasingly restrictive versions of $\mathcal{S}$.

(b) The system supports a cogent form of *dynamic program slicing* [KL88] as follows. Given a Maude program $\mathcal{M}$ and a trace slice $\mathcal{T}^\star$ for $\mathcal{M}$, *i*Julienne is able to infer the minimal fragment of $\mathcal{M}$ (i.e., the *program slice*) that is needed to reproduce $\mathcal{T}^\star$. This is done by uncovering statement dependences among computationally related parts of the program via backward trace slicing. This feature greatly facilitates the debugging of faulty Maude programs, since the user can generate a sequence of increasingly smaller program slices that gradually shrinks the area that contains the buggy piece of code.

(c) *i*Julienne is endowed with a powerful and intuitive Web user interface that allows the slicing criteria to be easily defined by either highlighting the chosen target symbols or by applying a user-defined filtering pattern. A browsing facility is also provided that enables forward and backward navigation through the trace (and the trace slice) and allows the user to examine all the information that is involved within each state transition (and its corresponding sliced counterpart) for debugging and comprehension purposes. The user interface can be tuned to provide distinct abstract views of the trace that aim at supporting different program comprehension levels. For instance, this includes hiding or displaying the auxiliary transformations that are used by Maude to handle associative and commutative operators.

Plan of the thesis

Following some preliminaries, the backward slicing technique presented in [ABFR12a, ABFR12b] is recalled in Chapter 1. In Chapter 2, we present *i*Julienne [ABFS13b], our online trace analyzer that implements the proposed slicing technique and we discuss the most relevant technical details together with some benchmarks that assess the practicality of the tool. Two different sessions working with the tool are illustrated also in this chapter. Finally, some conclusions and directions for future work are presented in the last chapter.

Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [BKdV03] and Rewriting Logic [Mes92]. Some familiarity with the Maude language [CDE+11] is also required.

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [CDE+11]. 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 x such that $w_1.x = w_2$. Given a set of positions P, the *prefix closure* of P is the set $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$. Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t|_w$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of *replacing the subterm* $t|_w$ by the term s.

A substitution σ is a mapping from variables to terms $\{X_1/t_1, \ldots, X_n/t_n\}$ such that $X_i\sigma = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq x_j$ if $i \neq j$), and $X\sigma = X$ for all other variables X. Given a substitution $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$, the *domain*

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

An *equational condition* b is an equation $t = t'$ with $t, t' \in \tau(\Sigma, \mathcal{V})$. This includes both, ordinary equations and abbreviated boolean equations $b = true$, with $b \in \tau(\Sigma, \mathcal{V})$ of sort Bool. A *matching condition* is a pair $p := e$ with $e, p \in \tau(\Sigma, \mathcal{V})$. A *rewrite expression* is a pair $e \Rightarrow p$, with $e, p \in \tau(\Sigma, \mathcal{V})$.

A *conditional* equation is an expression of the form $\lambda = \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is either an equational condition, or a matching condition. When the condition C is empty, we simply write $\lambda = \rho$. A conditional equation $\lambda = \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible*, iff (i) $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{i=1}^{n} \mathcal{V}ar(c_i)$, and (ii) for each c_i, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is an equational condition, and $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is a matching condition $p := e$.

A *conditional* rule is an expression of the form $\lambda \to \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is an equational condition, a matching condition, or a rewrite expression. When the condition C is empty, we simply write $\lambda \to \rho$. A conditional rule $\lambda \to \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible* iff it fulfils the exact analogous of the admissibility constraints (i) and (ii) for the equational conditions and the matching conditions, plus the following additional constraint: for each rewrite expression c_i in C of the form $e \Rightarrow p$, $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$.

The set of variables that occur in a (conditional) rule/equation r is denoted by $\mathcal{V}ar(r)$. Note that admissible equations and rules can contain extra-variables (i.e., variables that appear in the right-hand side or in the condition of a rule/equation but do not occur in the corresponding left-hand side). The admissibility requirements ensure that all the extra-variables will become instantiated whenever an admissible rule/equation is applied.

Conditional Rewriting Modulo Equational Theories

An *order-sorted equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of (oriented) admissible, conditional equations, and B is a collection of unconditional equational axioms (e.g., associativity, commutativity, and unity) that can be associated with any binary operator of Σ.

The equational theory E induces a congruence relation on the term algebra $T(\Sigma, \mathcal{V})$, which is denoted by $=_E$. A *conditional rewrite theory* (or simply, rewrite theory) is a triple $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where $(\Sigma, \Delta \cup B)$ is an order-sorted equational theory, and R is a set of admissible conditional rules[1].

Example B

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

```
mod BANK is inc INT .
  sorts Account Msg State Id .
  subsorts  Account Msg < State .

  var Id Id1 Id2 : Id .
  var bal bal1 bal2 newBal newBal1 newBal2 M : Nat .

  op empty-state : -> State .
  op _;_ : State State -> State [assoc comm id: empty-state] .
  op <_|_> : Id Nat -> Account [ctor] .
  ops credit debit : Id Nat -> Msg [ctor] .
  op  transfer : Id Id Nat -> Msg  [ctor] .

  crl [credit] : <Id|bal>;credit(Id,M) => <Id|newBal>
                 if newBal := bal + M .
  crl [debit] : <Id|bal>;debit(Id,M) => <Id|newBal>
                if bal >= M /\ newBal := bal - M .
  crl [transfer] : <Id1|bal1>;<Id2|bal2>;transfer(Id1,Id2,M) =>
                   <Id1|newBal1>;<Id2|newBal2>
                   if <Id1|bal1>;debit(Id1,M) => <Id1|newBal1>
                   /\ <Id2|bal2>;credit(Id2,M) => <Id2|newBal2> .
endm
```

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

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

Given a conditional rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the conditional rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual conditional rewrite relation on terms [Klo92] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [BM06], that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is in general undecidable, since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The conditional slicing technique formalized in this work is formulated by considering the precise way in which Maude proves the conditional rewriting steps (see Section 5.2 in [CDE+11]). Actually, the Maude interpreter implements conditional rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$, that allow rules and equations to be intermixed in the rewriting process by simply using an algorithm of matching modulo B. We define $\rightarrow_{R\cup\Delta,B}$ as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of Δ (oriented from left to right) as simplification rules: thus, for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a term $t\!\downarrow_\Delta$ to which no further equations can be applied. The term $t\downarrow_\Delta$ is called a *canonical form* of t w.r.t. Δ. On the other hand, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [CDE+11].

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

Note that the evaluation of a condition C is typically a recursive process,

since it may involve further (conditional) rewrites in order to normalize C to *true*. Specifically, an equational condition e *evaluates to true* w.r.t. σ if $e\sigma{\downarrow_\Delta}{=_B}\ true$; a matching equation $p := t$ *evaluates to true* w.r.t. σ if $p\sigma =_B t\sigma{\downarrow_\Delta}$; a rewrite expression $t \Rightarrow p$ *evaluates to true* w.r.t. σ if there exists a rewrite sequence $t\sigma \to^*_{R\cup\Delta,B} u$, such that $u =_B p\sigma$[2]. Although rewrite expressions and matching/equational conditions can be intermixed in any order, we assume that their satisfaction is attempted sequentially from left to right, as in Maude.

Under appropriate conditions on the rewrite theory, a rewrite step modulo E on a term t can be implemented without loss of completeness by applying the following rewrite strategy [DM10]:

(i) reduce t w.r.t. $\to_{\Delta,B}$ until the canonical form $t\downarrow_\Delta$ is reached;

(ii) rewrite $t\downarrow_\Delta$ w.r.t. $\to_{R,B}$.

An *execution trace* $\mathcal{T}$ in the rewrite theory $(\Sigma, \Delta \cup B, R)$ is a rewrite sequence $s_0 \to^*_{\Delta,B} s_0{\downarrow_\Delta} \to_{R,B} s_1 \to^*_{\Delta,B} s_1{\downarrow_\Delta} \ldots$ that interleaves $\to_{\Delta,B}$ rewrite steps and $\to_{R,B}$ steps following the strategy mentioned above.

Given an execution trace $\mathcal{T}$, it is always possible to expand $\mathcal{T}$ in an *instrumented* trace $\mathcal{T}'$ in which every application of the matching modulo B algorithm is mimicked by the explicit application of a suitable equational axiom, which is also oriented as a rewrite rule [ABER11]. This way, any given instrumented execution trace consists of a sequence of (standard) rewrites using the conditional equations ($\to_\Delta$), conditional rules ($\to_R$), and axioms ($\to_B$).

Example C

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

[2]Technically, to properly evaluate a rewrite expression $t \Rightarrow p$ or a matching condition $p := t$, the term p is required to be a Δ-pattern —i.e., a term p such that, for every substitution σ, if $x\sigma$ is a canonical form w.r.t. Δ for every $x \in Dom(\sigma)$, then $p\sigma$ is also a canonical form w.r.t. Δ.

Also, typically hidden inside the B-matching algorithms, some transformations allow terms that contain operators that obey associative-commutative axioms to be rewritten by first producing a single representative of their AC congruence class [ABER11]. For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b, f(f(b, a), c)) =_B f(f(b, c), f(a, b))$, we can represent it by using the "internal sequence" of transformations $f(b, f(f(b, a), c)) \rightarrow^*_{flat_B} f(a, b, b, c) \rightarrow^*_{unflat_B} f(f(b, c), f(a, b))$, where the first one corresponds to a *flattening* transformation sequence that obtains the AC canonical form, while the second one corresponds to the inverse, unflattening one.

In the sequel, we assume all execution traces are instrumented as explained above. By abuse of notation, we frequently denote the rewrite relations $\rightarrow_\Delta$, $\rightarrow_R$, $\rightarrow_B$ by $\rightarrow$. Also, by $\rightarrow^*$ (resp. $\rightarrow^+$), we denote the transitive and reflexive (resp. transitive) closure of the relation $\rightarrow_\Delta \cup \rightarrow_R \cup \rightarrow_B$.

1

Backward Trace Slicing for Conditional Rewrite Theories

In this chapter, we recall the backward conditional slicing algorithm for RWL computations of [ABFR12a]. The algorithm is formalized by means of a transition system that traverses the execution traces from back to front. The transition system is given by a single inference rule that relies on a *backward rewrite step slicing* procedure that is based on substitution refinement.

This chapter is organized as follows. Section 1.1, introduces the notions of term slice and term slice concretization. Section 1.2 describes the backbone of the backward trace slicing technique for RWL computation traces. Section 1.3 formalizes the slicing algorithm for single execution steps and finally, Section 1.4 reports the experimental evaluation of the slicing technique.

1.1 Term slices and term slice concretizations

A term slice of a term t is a term abstraction that disregards part of the information in t, that is, the irrelevant data in t are simply replaced by special $\bullet$-variables, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$, which are generated by calling the auxiliary function $\mathit{fresh}^{\bullet}$[1]. More formally, a term slice is defined as follows.

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

Definition 1.1.1 (term slice) *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term, and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. A* term slice *of t w.r.t. P is defined as follows:*

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

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

When P is understood, a term slice of t w.r.t. P is simply denoted by $t^{\bullet}$.

Roughly speaking, a term slice t w.r.t. a set of positions P includes all symbols of t that occur within the paths from the root to any position in P, while each maximal subterm $t_{|p}$, with $p \notin P$, is abstracted by means of a $\bullet$-variable.

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

Example 1.1.2

Let $t = d(f(g(a, h(b)), c), a)$ be a term, and let $P = \{1.1,\ 1.2\}$ be a set of positions of t. By applying Definition 1.1.1, we get the term slice $t^{\bullet} = slice(t, P) = d(f(g(\bullet_1, \bullet_2), y), \bullet_3)$ and the set of meaningful positions $\mathcal{MP}os(t^{\bullet}) = \{\Lambda,\ 1,\ 1.1,\ 1.2\}$.

To particularize a term slice, $\bullet$-variables must be instantiated with data that satisfy a given boolean condition called *compatibility* condition. Term slice concretization is formally defined as follows.

Definition 1.1.3 (term slice concretization) *Let $t, t' \in \tau(\Sigma, \mathcal{V})$ be two terms. Let $t^{\bullet}$ be a term slice of t and let $B^{\bullet}$ be a boolean condition. We say that t' is a* concretization *of $t^{\bullet}$ that is compatible with $B^{\bullet}$ (in symbols $t^{\bullet} \propto^{B^{\bullet}} t'$), if (i) there exists a substitution σ such that $t^{\bullet}\sigma = t'$, and (ii) $B^{\bullet}\sigma$ evaluates to true.*

Example 1.1.4

Let $t^{\bullet} = \bullet_1 + \bullet_2 + \bullet_2$ and $B^{\bullet} = (\bullet_1 > 6 \wedge \bullet_2 \leq 7)$. Then, $10 + 2 + 2$ is a concretization of $t^{\bullet}$ that is compatible with $B^{\bullet}$, while $4 + 2 + 2$ is not.

In the following, a backward trace slicing technique is formulated. Given an execution trace $\mathcal{T} : s_0 \rightarrow^* s_n$ and a term slice $s_n^\bullet$ of s_n, this algorithm generates the sliced counterpart $\mathcal{T}^\bullet : s_0^\bullet \rightarrow^* s_n^\bullet$ of $\mathcal{T}$ that only encodes the information required to reproduce (the meaningful symbols of) the term slice $s_n^\bullet$. Additionally, the algorithm returns a companion compatibility condition $B^\bullet$ that guarantees the correctness of the generated trace slice.

1.2 Backward Slicing for Execution Traces

Consider an execution trace $\mathcal{T} : s_0 \rightarrow^* s_n$. A trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is defined w.r.t. a *slicing criterion* — i.e., a set of positions $\mathcal{O}_{s_n} \subseteq \mathcal{P}os(s_n)$ that refer to those symbols of s_n that we want to observe. Basically, the trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is obtained by removing all the information from $\mathcal{T}$ that is not required to produce the term slice $s_n^\bullet = slice(s_n, \mathcal{O}_{s_n})$. A trace slice is formally defined as follows.

Definition 1.2.1 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} s_1 \overset{r_2,\sigma_2,w_2}{\rightarrow} \ldots \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. A* trace slice *of $\mathcal{T}$ w.r.t. $\mathcal{O}_{s_n}$ is a pair $[s_0^\bullet \rightarrow s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B^\bullet]$, where*

1. *$s_i^\bullet$ is a term slice of s_i, for $i = 0, \ldots, n$, and $B^\bullet$ is a boolean condition;*
2. *$s_n^\bullet = slice(s_n, \mathcal{O}_{s_n})$;*
3. *for every term s_0' such that $s_0^\bullet \propto^{B^\bullet} s_0'$, there exists an execution trace $s_0' \rightarrow s_1' \rightarrow \ldots \rightarrow s_n$ in $\mathcal{R}$ such that*
   - *i) $s_i' \rightarrow s_{i+1}'$ is either the rewrite step $s_i' \overset{r_{i+1},\sigma_{i+1}',w_{i+1}}{\rightarrow} s_{i+1}'$ or $s_i' = s_{i+1}'$, $i = 0, \ldots, n-1$;*
   - *ii) $s_i^\bullet \propto^{B^\bullet} s_i'$, $i = 1, \ldots, n$.*

Note that Point 3 of Definition 1.2.1 ensures that the rules involved in the sliced steps of $\mathcal{T}^\bullet$ can be applied again, at the corresponding positions, to every concrete trace $\mathcal{T}'$ that can be obtained by instantiating all the $\bullet$-variables in $s_0^\bullet$ with arbitrary terms.

The following example illustrates the slicing of an execution trace.

Example 1.2.2

Consider the Maude specification of Example B together with the following execution trace $\mathcal{T}$:

$$\texttt{(<a|30>;debit(a,5));credit(a,3)} \overset{\texttt{debit}}{\rightarrow} \texttt{<a|25>;credit(a,3)} \overset{\texttt{credit}}{\rightarrow} \texttt{<a|28>}$$

Let $\texttt{<a|}\bullet_1\texttt{>}$ be a term slice of <a|28> generated with the slicing criterion {1} —i.e., $\texttt{<a|}\bullet_1\texttt{>}$=*slice*(<a|28>, {1}). Then, the trace slice for $\mathcal{T}$ is $[\mathcal{T}^\bullet,\ \bullet_8 \geq \bullet_9]$ where $\mathcal{T}^\bullet$ is as follows:

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

Note that $\mathcal{T}^\bullet$ needs to be endowed with the compatibility condition $\bullet_8 \geq \bullet_9$ in order to ensure the applicability of the `debit` rule. In other words, any instance $s^\bullet\sigma$ of $\texttt{<a|}\bullet_8\texttt{>;debit(a,}\bullet_9\texttt{)}$ can be rewritten by the `debit` rule only if $\bullet_8\sigma \geq \bullet_9\sigma$.

Informally, given a slicing criterion $\mathcal{O}_{s_n}$ for the execution trace $\mathcal{T} = s_0 \rightarrow^* s_n$, at each rewrite step $s_{i-1} \rightarrow s_i$, $i = n, \ldots, 1$, our technique inductively computes the association between the meaningful information of s_i and the meaningful information in s_{i-1}. For each such rewrite step, the conditions of the applied rule are recursively processed in order to ascertain from s_i the meaningful information in s_{i-1}, together with the accumulated condition $B^\bullet_i$. The technique proceeds backwards, from the final term s_n to the initial term s_0. A simplified trace is obtained where each s_i is replaced by the corresponding term slice $s^\bullet_i$.

We define a transition system $(\mathit{Conf},\ \bullet\!\!\rightarrow)$ [Plo04] where *Conf* is a set of *configurations* and $\bullet\!\!\rightarrow$ is the transition relation that implements the backward trace slicing algorithm. Configurations are formally defined as follows.

Definition 1.2.3 *A configuration, written as* $\langle \mathcal{T},\ S^\bullet,\ B^\bullet \rangle$*, consists of three components:*

- *the execution trace* $\mathcal{T} : s_0 \rightarrow^* s_{i-1} \rightarrow s_i$ *to be sliced;*
- *the term slice* $s^\bullet_i$*, that records the computed term slice of* s_i
- *a boolean condition* $B^\bullet$*.*

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

Definition 1.2.4 *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory, let* $\mathcal{T} = U \rightarrow^* W$ *be an execution trace in* $\mathcal{R}$*, and let* $V \rightarrow W$ *be a rewrite step. Let* $B^\bullet_W$

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

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

Roughly speaking, the relation $\bullet\!\!\rightarrow$ transforms a configuration $\langle U \to^* V \to W,\ W^\bullet,\ B_W^\bullet\rangle$ into a configuration $\langle U \to^* V,\ V^\bullet,\ B_V^\bullet\rangle$ by calling the function *slice-step*$(V \to W,\ W^\bullet,\ B_W^\bullet)$ of Section 1.3, which returns a rewrite step slice for $V \to W$. More precisely, *slice-step* computes a suitable term slice $V^\bullet$ of V and a boolean condition $B_V^\bullet$ that updates the compatibility condition specified by $B_W^\bullet$.

The initial configuration $\langle s_0 \to^* s_n,\ \mathit{slice}(s_n, \mathcal{O}_{s_n}),\ \mathit{true}\rangle$ is transformed until a terminal configuration $\langle s_0,\ s_0^\bullet,\ B_0^\bullet\rangle$ is reached. Then, the computed trace slice is obtained by replacing each term s_i by the corresponding term slice $s_i^\bullet$, $i = 0, \ldots, n$, in the original execution trace $s_0 \to^* s_n$. The algorithm additionally returns the accumulated compatibility condition $B_0^\bullet$ attained in the terminal configuration.

More formally, the backward trace slicing of an execution trace w.r.t. a slicing criterion is implemented by the function *backward-slicing* defined as follows.

Definition 1.2.5 (Backward trace slicing algorithm) *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory, and let* $\mathcal{T} : s_0 \to^* s_n$ *be an execution trace in* $\mathcal{R}$*. Let* $\mathcal{O}_{s_n}$ *be a slicing criterion for* $\mathcal{T}$*. Then, the function backward-slicing is computed as follows:*

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

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

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

In the following, we formulate the auxiliary procedure for the slicing of conditional rewrite steps.

1.3 The function *slice-step*

The function *slice-step*, which is outlined in Figure 1.1, takes as input three parameters, namely, a rewrite step $\mu : s \stackrel{r,\sigma,w}{\to} t$ (with $r = \lambda \to \rho \mathit{\ if\ } C$[2]), a term slice $t^\bullet$ of t, and a compatibility condition $B^\bullet_{prev}$; and delivers the term slice $s^\bullet$ and a new compatibility condition $B^\bullet$. Within the algorithm *slice-step*, we use an auxiliary operator $\langle\!| \sigma_1, \sigma_2 |\!\rangle$ that refines (overrides) a substitution σ_1 with a substitution σ_2, where both σ_1 and σ_2 may contain $\bullet$-variables. The main idea behind $\langle\!| \_, \_ |\!\rangle$ is that, for the slicing of the step μ, all variables in the applied rewrite rule r are naïvely assumed to be initially bound to irrelevant data $\bullet$, and the bindings are incrementally refined as we (partially) solve the conditions of r.

function *slice-step*$(s \stackrel{r,\sigma,w}{\to} t,\ t^\bullet,\ B^\bullet_{prev})$
if $w \notin \mathcal{MP}os(t^\bullet)$
then
  $s^\bullet = t^\bullet$
  $B^\bullet = B^\bullet_{prev}$
else
  $\theta = \{x/\mathit{fresh}^\bullet \mid x \in Var(r)\}$
  $\rho^\bullet = \mathit{slice}(\rho, \mathcal{MP}os(t^\bullet_{|w}))$
  $\psi_\rho = \langle\!| \theta, \mathit{match}_{\rho^\bullet\theta}(t^\bullet_{|w}) |\!\rangle$
  for $i = n$ **downto** 1 **do**
    $(\psi_i,\ B^\bullet_i) = \mathit{process\text{-}condition}(c_i,\ \sigma,\ \langle\!| \psi_\rho, \psi_n ... \psi_{i+1} |\!\rangle)$
  od
  $s^\bullet = t^\bullet[\lambda \langle\!| \psi_\rho, \psi_n ... \psi_1 |\!\rangle]_w$
  $B^\bullet = (B^\bullet_{prev} \wedge B^\bullet_n ... \wedge B^\bullet_1)(\psi_1 \psi_2 \ldots \psi_n)$
fi
return $(s^\bullet,\ B^\bullet)$

Figure 1.1: Backward step slicing function.

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

Definition 1.3.1 (refinement) *Let σ_1 and σ_2 be two substitutions. The refinement of σ_1 w.r.t. σ_2 is defined by the operator $\langle\!|\_,\_|\!\rangle$ as follows: $\langle\!|\sigma_1, \sigma_2|\!\rangle = \sigma_{\restriction Dom(\sigma_1)}$, where*

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

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

Example 1.3.2

Let $\sigma_1 = \{x/\bullet_1,\ y/\bullet_2\}$ and $\sigma_2 = \{x/a,\ \bullet_2/g(\bullet_3),\ z/5\}$ be two substitutions. Thus, $\langle\!|\sigma_1, \sigma_2|\!\rangle = \{x/a,\ y/g(\bullet_3)\}$.

Roughly speaking, the function *slice-step* works as follows. When the rewrite step μ occurs at a position w that is not a meaningful position of $t^\bullet$ (in symbols, $w \notin \mathcal{MP}os(t^\bullet)$), trivially μ does not contribute to producing the meaningful symbols of $t^\bullet$. Therefore, the function returns $s^\bullet = t^\bullet$, with the input compatibility condition $B^\bullet_{prev}$.

Example 1.3.3

Consider the Maude specification of Example B and the following rewrite step μ: `(<a|30>;debit(a,5));credit(a,3)` $\overset{\texttt{debit}}{\rightarrow}$ `<a|25>;credit(a,3)`. Let $\bullet_1$;`credit`(a, 3) be a term slice of `<a|25>;credit(a,3)`. Since the rewrite step μ occurs at position 1 $\not\in \mathcal{MP}os(\bullet_1;\texttt{credit}(\texttt{a},3))$, the term `<a|25>` introduced by μ in `<a|25>;credit(a,3)` is completely ignored in $\bullet_1$;`credit`(a, 3). Hence, the computed term slice for `(<a|30>;debit(a,5)); credit(a,3)` is the very same $\bullet_1$;`credit`(a, 3).

On the other hand, when $w \in \mathcal{MP}os(t^\bullet)$, the computation of $s^\bullet$ and $B^\bullet$ involves a more in-depth analysis of the rewrite step, which is based on an inductive refinement process that is obtained by recursively processing the conditions of the applied rule. More specifically, we initially define the substitution $\theta = \{x/\mathit{fresh}^\bullet \mid x \in Var(r)\}$ that binds each variable in r to a fresh $\bullet$-variable. This corresponds to assuming that all the information in μ, which is introduced by the substitution σ, can be marked as irrelevant. Then, θ is incrementally refined using the following two-step procedure.

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

Figure 1.2: Condition processing function.

Step 1. We compute the matcher $match_{\rho\theta}(t^\bullet_{|w})$, and then generate the refinement ψ_ρ of θ w.r.t. $match_{\rho\theta}(t^\bullet_{|w})$ (in symbols, $\psi_\rho = \langle\!|\theta, match_{\rho\theta}(t^\bullet_{|w})|\!\rangle$). Roughly speaking, the refinement ψ_ρ updates the bindings of θ with the meaningful information extracted from $t^\bullet_{|w}$.

Example 1.3.4

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

Step 2. Let $C\sigma = c_1\sigma \wedge \ldots \wedge c_n\sigma$ be the instance of the condition in the rule r that enables the rewrite step μ. Each (sub)condition $c_i\sigma$, $i =$

$1, \ldots, n$ is processed, in reversed evaluation order, i.e., from $c_n\sigma$ to $c_1\sigma$, by using the auxiliary function *process-condition* given in Figure 1.2 that generates a pair $(\psi_i,\ B_i^\bullet)$ such that ψ_i is used to further refine the partially ascertained substitution $\langle\!|\psi_\rho, \psi_n, \ldots, \psi_{i+1}|\!\rangle$ computed by incrementally analyzing conditions $c_n\sigma, c_{n-1}\sigma \ldots, c_{i+1}\sigma$, and $B_i^\bullet$ is a boolean condition that is derived from the analysis of the condition c_i.

When the whole $C\sigma$ has been processed, we get the refinement $\langle\!|\psi_\rho, \psi_n, \ldots, \psi_1|\!\rangle$, which basically encodes all the instantiations required to construct the term slice $s^\bullet$ from $t^\bullet$. More specifically, $s^\bullet$ is obtained from $t^\bullet$ by replacing the subterm $t^\bullet_{|w}$ with the left-hand side λ of r instantiated with $\langle\!|\psi_\rho, \psi_n, \ldots, \psi_1|\!\rangle$. Furthermore, $B^\bullet$ is built by collecting all the boolean compatibility conditions $B_i^\bullet$ delivered by *process-condition* and instantiating them with the composition of the computed refinements $\psi_1 \ldots \psi_n$. It is worth noting that *process-condition* handles rewrite expressions, equational conditions, and matching conditions differently. More specifically, the pair $(\psi_i,\ B_i)$ that is returned after processing each condition c_i is computed as follows.

– **Matching conditions.** Let c be a matching condition with the form $p := m$ in the condition of rule r. During the execution of the step $\mu : s \overset{r,\sigma,w}{\rightarrow} t$, recall that c is evaluated as follows: first, $m\sigma$ is reduced to its canonical form $m\sigma\downarrow_\Delta$, and then the condition $m\sigma\downarrow_\Delta =_B p\sigma$ is checked. Therefore, the analysis of the matching condition $p := m$ during the slicing process of μ implies slicing the (internal) execution trace $\mathcal{T}_{int} = m\sigma \rightarrow^* p\sigma$, which is done by recursively invoking the function *backward-slicing* for execution trace slicing with respect to the meaningful positions of the term slice $p\theta$ of p, where θ is a refinement that records the meaningful information computed so far. That is, $[m^\bullet \rightarrow^* p^\bullet,\ B^\bullet] = \textit{backward-slicing}(m\sigma \rightarrow^* p\sigma,\ \mathcal{MP}os(p\theta))$. The result delivered by the function *backward-slicing* is a trace slice $m^\bullet \rightarrow^* p^\bullet$ with compatibility condition $B^\bullet$.

  In order to deliver the final outcome for the matching condition $p := m$, first the substitution $\psi = match_{m\theta}(m^\bullet)$ is computed, which is the substitution needed to refine θ, is computed, and then the pair $(\psi,\ B^\bullet)$ is returned.

Example 1.3.5

Consider the the rewrite step $\mu_{\texttt{debit}}$ of Example 1.3.4 together with the refined substitution $\theta = \{\texttt{Id/a},\ \texttt{bal/}\bullet_3,\ \texttt{M/}\bullet_4,\ \texttt{newBal/}\bullet_1\}$. We

process the condition `newBal := bal - M` of `debit` by considering the internal execution trace $\mathcal{T}_{int} = \texttt{30} - \texttt{5} \rightarrow \texttt{25}$ [3]. By invoking the function *backward-slicing*, the trace slice result is $[\bullet_6 \rightarrow \bullet_6,\ true]$. The final outcome is given by $match_{\bullet_7 - \bullet_8}(\bullet_6)$, that is $fail$. Thus, we conclude that θ does not need any further refinement.

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

– **Equational conditions.** During the execution of the rewrite step $\mu : s \overset{r,\sigma,w}{\rightarrow} t$, the instance $e\sigma$ of an equational condition e in the condition of the rule r is just fulfilled or falsified, but it does not bring any instantiation into the output term t. Therefore, when processing $e\sigma$, no meaningful information to further refine the partially ascertained substitution θ must be added. However, the equational condition e must be recorded in order to compute the compatibility condition $B^\bullet$ for the considered conditional rewrite step. In other words, after processing an equational condition e, we deliver the tuple $(\psi,\ B^\bullet)$, with $\psi = \{\ \}$ and $B^\bullet = e\theta$. Note that the condition e is instantiated with the updated substitution θ, in order to transfer only the meaningful information of $e\sigma$ computed so far in e.

Example 1.3.6

Consider the refined substitution given in Example 1.3.5 $\theta = \{\texttt{Id/a},\ \texttt{bal}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{newBal}/\bullet_1\}$ together with the rewrite step $\mu_{\texttt{debit}}$ of Example 1.3.4 that involves the application of the `debit` rule. After processing the condition `bal >= M` of `debit`, we deliver $B^\bullet = (\bullet_3 >= \bullet_4)$.

Correctness of our conditional slicing technique is established by the following theorem. The proof can be found in [ABFR12c].

[3] Note that the trace `30-5→25` involves an application of the Maude built-in operator "`-`". Given a built-in operator `op`, in order to handle the reduction `a op b` → `c` as an ordinary rewrite step, we add the rule `a op b` ⇒ `c` to the considered rewrite theory.

Theorem 1.3.7 (correctness) *Let $\mathcal{R}$ be a rewrite theory. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} ... \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ be an execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \rightarrow ... \rightarrow s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing($\mathcal{T}$, $\mathcal{O}_{s_n}$) is a trace slice for $\mathcal{T}$.*

1.4 Experimental Evaluation

The conditional slicing methodology presented so far has been implemented in Julienne, a prototype tool written in Maude first described in [ABFR12b] and publicly available at `http://safe-tools.dsic.upv.es/julienne/`. The tool takes in input a slicing criterion and a Maude execution trace, which is a term of sort `Trace` (generated by means of the the Maude meta-level operator `metaSearchPath`), and delivers the corresponding trace slice. The prototype have been tested on rather large execution traces, such as the counterexamples generated by the model checker for Web applications Web-TLR [ABER10]. In our experiments, we have considered a Webmail application together with four LTLR properties that have been refuted by Web-TLR. For each refuted property, Web-TLR has produced the corresponding counterexample in the form of a huge, textual execution trace $\mathcal{T}_i$, $i = 1, ..., 4$, in the range $10 - 100Kb$ that has been used to feed our slicer.

Example trace	**Original trace size**	**Slicing criterion**	**Sliced trace size**	**% *reduction***
Web-TLR.$\mathcal{T}_1$	19114	Web-TLR.$\mathcal{T}_1$.O_1	3982	79.17%
		Web-TLR.$\mathcal{T}_1$.O_2	3091	83.83%
Web-TLR.$\mathcal{T}_2$	22018	Web-TLR.$\mathcal{T}_2$.O_1	2984	86.45%
		Web-TLR.$\mathcal{T}_2$.O_2	2508	88.61%
Web-TLR.$\mathcal{T}_3$	38983	Web-TLR.$\mathcal{T}_3$.O_1	2045	94.75%
		Web-TLR.$\mathcal{T}_3$.O_2	2778	92.87%
Web-TLR.$\mathcal{T}_4$	69491	Web-TLR.$\mathcal{T}_4$.O_1	8493	87.78%
		Web-TLR.$\mathcal{T}_4$.O_2	5034	92.76%

Table 1.1: Backward trace slicing benchmarks.

Table 1.1 shows the size of the original counterexample trace and that of the computed trace slice, both measured as the length of the corresponding string, w.r.t. two slicing criteria, that are detailed in the tool website. The

considered criteria allow one to monitor the messages exchanged by a specific Web browser and the Webmail server, as well as to isolate the changes on the data structures of the two interacting entities. The *%reduction* column in Table 1.1 refers to the percentage of reduction achieved. The results we have obtained are very encouraging, and show an impressive reduction rate (up to $\sim$ 95%) in reasonable time (max. 0.9s on a Linux box equipped with an Intel Core 2 Duo 2.26GHz and 4Gb of RAM memory). Actually, sometimes the trace slices are small enough to be easily inspected by the users, who can restrict their attention to the part of the computation that they want to observe.

2

Slicing-based Trace Analysis of Rewriting Logic Specifications with *i*Julienne

In this chapter, we present an incremental slicing methodology specially suitable for the analysis of rewriting logic computations together with its implementation in the *i*Julienne tool. This methodology is a new extended version of the original slicing technique first proposed in [ABFR12a].

This chapter is organized as follows. Section 2.1 gives the intuition of our incremental backward trace slicing methodology. Section 2.2 presents *i*Julienne, the slicing-based trace analyzer implementing this new methodology. Section 2.3 shows two different working sessions with *i*Julienne and finally, Section 2.4 reports some experiments that assess the practicality of our methodology.

2.1 Incremental Trace Slicing

The original trace slicing technique of [ABFR12a] intuitively works as shown in Figure 2.1. Given a starting slicing criterion (i.e., a piece of relevant information that we want to observe in the final computation state), the trace is traversed from back to front and the backward dependence of the observed information is incrementally computed at each execution step.

The information computed at each step is surely relevant with respect to the slicing criterion. However, this information may not be relevant with respect to the source of error we want to discover, as it may contain extra irrelevant information in this regard. Many factors can influence this excess of

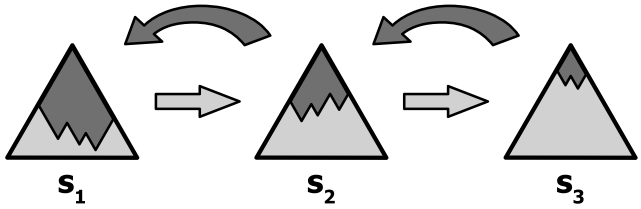

Figure 2.1: Backward trace slicing scheme.

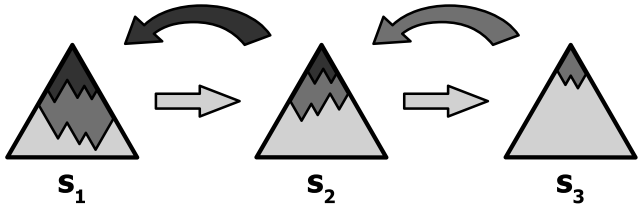

Figure 2.2: Incremental backward trace slicing scheme.

information: an erroneous or overly general slicing criterion, the structure of the data dependence itself, the length of the trace (as the computed relevant information tends to grow at each execution step), etc.

In order to reduce to the minimum the excess of irrelevant information with respect to the source of error and therefore achieve better reduction rates, we propose to refine the slicing criterion at each execution step by means of the auxiliary, interactive function $refine$, which takes as input a slicing criterion and returns an equal or more restrictive one.

The proposal, which is informally depicted in Figure 2.2, works as follows. Given an execution trace $\mathcal{T} : s_0 \rightarrow^* s_n$ and a slicing criterion $\mathcal{O}_{s_n} \subseteq \mathcal{P}os(s_n)$, which refers to those symbols of s_n that we want to observe, the main difference between the proposed incremental slicing methodology and the original backward slicing technique is that a refined version of the slicing criterion is established after each execution of the *slice-step* function.

Consider the configuration given in Definition 1.2.3 and the transition system $(Conf, \bullet\!\!\rightarrow)$ given in Definition 1.2.4. In order to endow incremental capabilities to the original backward trace slicing technique, we modify the slicing calculus as follows.

Definition 2.1.1 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory, let $\mathcal{T} = U \to^* W$ be an execution trace in $\mathcal{R}$, and let $V \to W$ be a rewrite step. Let $B^\bullet_W$ and $B^\bullet_V$ be two boolean conditions, and let $W^\bullet$ be a term slice of W. Then, the transition relation $\bullet\!\!\to \subseteq Conf \times Conf$ is the smallest relation that satisfies the following rule:*

$$\frac{(V^\bullet,\ B^\bullet_V) = \mathit{slice\text{-}step}(V \to W,\ W^{\bullet\prime},\ B^\bullet_W) \quad \wedge\ W^{\bullet\prime} = \mathit{slice}(W,\ \mathit{refine}(\mathcal{MP}os(W^\bullet)))}{\langle U \to^* V \to W,\ W^\bullet,\ B^\bullet_W\rangle \bullet\!\!\to \langle U \to^* V,\ V^\bullet,\ B^\bullet_V\rangle}$$

Roughly speaking, the relation $\bullet\!\!\to$ transforms a configuration $\langle U \to^* V \to W,\ W^\bullet,\ B^\bullet_W\rangle$ into a configuration $\langle U \to^* V,\ V^\bullet,\ B^\bullet_V\rangle$ by calling the function $\mathit{slice\text{-}step}(V \to W,\ W^{\bullet\prime},\ B^\bullet_W)$ of Section 1.3, which returns a rewrite step slice for $V \to W$, and by allowing the user to refine the meaningful symbols of $W^\bullet$ to calculate $W^{\bullet\prime}$ as the $\mathit{slice}(W,\ \mathit{refine}(\mathcal{MP}os(W^\bullet)))$. More precisely, *slice-step* computes a suitable term slice $V^\bullet$ of V and a compatibility condition B_V that updates the compatibility condition specified by B_W, and *refine* computes a refined slicing criterion, which is used to slice W and therefore obtain the new slicing criterion as the meaningful positions of $W^{\bullet\prime}$.

In the same way as the original definition, the initial configuration $\langle s_0 \to^* s_n,\ \mathit{slice}(s_n, \mathcal{O}_{s_n}),\ \mathit{true}\rangle$ is transformed until a terminal configuration $\langle s_0,\ s^\bullet_0,\ B^\bullet_0\rangle$ is reached. Then, the computed trace slice is obtained by replacing each term s_i by the corresponding term slice $s^\bullet_i$, $i = 0, \ldots, n$, in the original execution trace $s_0 \to^* s_n$. The algorithm additionally returns the accumulated compatibility condition $B^\bullet_0$ attained in the terminal configuration.

More formally, the incremental backward trace slicing of an execution trace with respect to a slicing criterion is implemented by the function *incremental-backward-slicing*, defined as follows.

Definition 2.1.2 (Incremental backward trace slicing algorithm) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory, and let $\mathcal{T} : s_0 \to^* s_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. Then, the function incremental-backward-slicing is computed as follows:*

$$\mathit{incremental\text{-}backward\text{-}slicing}(s_0 \to^* s_n, \mathcal{O}_{s_n}) = [s^\bullet_0 \to^* s^\bullet_n,\ B^\bullet_0]$$

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

$$\langle s_0 \to^* s_n,\ s^\bullet_n,\ \mathit{true}\rangle \bullet\!\!\to \langle s_0 \to^* s_{n-1},\ s^\bullet_{n-1},\ B^\bullet_{n-1}\rangle \bullet\!\!\to^* \langle s_0,\ s^\bullet_0,\ B^\bullet_0\rangle$$
$$\text{where } s^\bullet_n = \mathit{slice}(s_n,\ \mathcal{O}_{s_n})$$

2.2 The *i*Julienne Online Trace Analyzer

In this section we present *i*Julienne, the online trace analyzer that implements the incremental backward trace slicing methodology presented so far.

2.2.1 Features and Characteristics of *i*Julienne

The *i*Julienne online trace analyzer provides the means for the user to perform exhaustive analyses of Maude execution traces, also including a trace generation facility. The main features *i*Julienne offers to perform such analyses are summarized as follows:

i) Trace generation capabilities by providing an initial and final state of the desired trace, either in source-level or meta-level representation.

ii) Ability to perform an exhaustive analysis of each rule or equation application via navigating the states slider and obtaining useful information related to each transition, that includes the rule or equation applied, the computed substitution and the position where they were applied.

iii) Integration of a simple but highly effective *querying language* to perform elaborated queries in search for relevant patterns in the entire trace.

iv) Highly customizable incremental backward trace slicing feature, which allows the user to, either refine the current slicing criterion, or specify a completely new one in any state of the trace.

v) Supports a cogent form of *Dynamic Program Slicing*, eliding irrelevant operators, rules and equations from the provided Maude program with respect to a previously performed trace slice.

By combining all these features, the user can highly reduce the time and effort required to perform comprehensive analyses of Maude execution traces, which is particularly suitable as programs and traces grow in size and complexity.

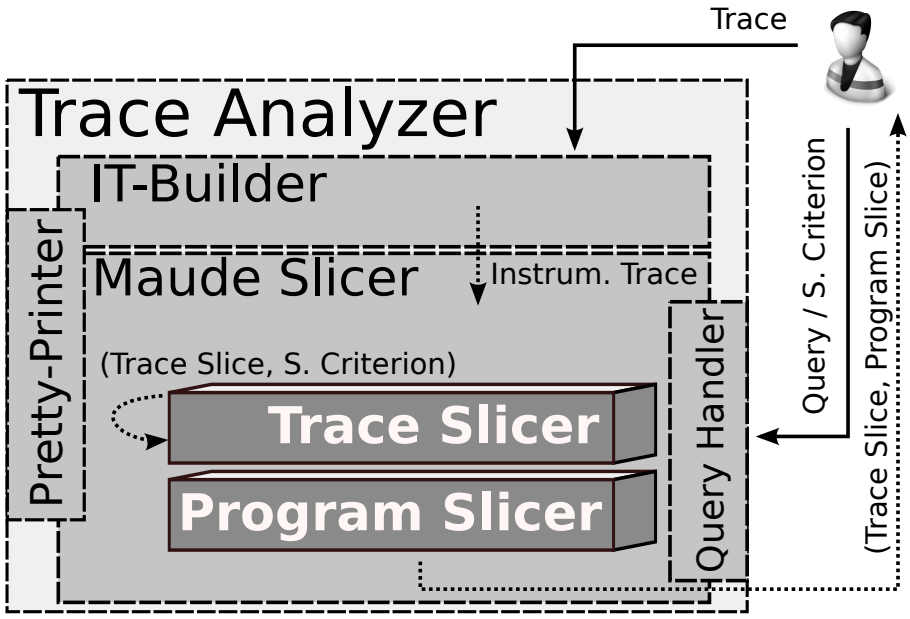

Figure 2.3: *i*Julienne architecture

2.2.2 The System Architecture of *i*Julienne

The *i*Julienne system is written in Maude and consists of about 250 Maude function definitions (approximately 1.7K lines of source code). It is a stand-alone application (which can be invoked as a Full Maude command or used online through a Java Web service) that allows the analysis of general rewrite theories that may contain (conditional) rules and equations, built-in operators, and algebraic axioms.

The user interface of *i*Julienne is based on the AJAX technology, which allows the Maude engine to be used through the WWW. The tool is publicly available at `http://safe-tools.dsic.upv.es/iJulienne`. Its architecture, which is depicted in Figure 2.3, consists of four main modules named **IT-Builder**, **Maude Slicer**, **Query Handler**, and **Pretty-Printer**.

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

The **Maude Slicer** module provides incremental trace slicing and dynamic program slicing facilities. Both of these techniques are developed by using Maude reflection and meta-level functionality. On one hand, the *Trace*

Slicer implements a greatly enhanced, incremental extension of the conditional backward trace slicing algorithm of [ABER11, ABFR12a] in which the slicing criteria can be repeatedly refined and the corresponding trace slices are automatically obtained by simply discarding the pieces of information affected by the updates. Thanks to incrementality, both trace slicing and analysis and debugging times are significantly reduced. On the other hand, the companion *Program Slicer* can be used to discard the program equations and rules that are not responsible for producing the set of target symbols in the observed trace state. Rather than simply glueing together the program equations and rules that are used in the simplified trace, it just delivers the minimal program fragment that is proved to influence the observed result. In other words, not only are the unused program data and rules removed but the data and rules that are used in sub-computations that are irrelevant to the criterion of interest are also removed.

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

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

2.2.3 Trace Querying

Execution traces (and in particular Maude execution traces) commonly consist of a high number of huge and complex states. In this regard, the possibility of querying about the trace in order to properly analyze their contained information is a must-have feature for any practical debugging tool. *i*Julienne offers this possibility by implementing a querying mechanism

based on a simple but highly effective pattern-matching filtering language that is particularly useful for both trace analysis and program debugging by allowing the trace to be queried and the slicing criterion to be fined incrementally.

One of the most relevant characteristics about our *querying* language is the use of two symbols as wild-cards, namely `?` and _, that are associated with the actions of identification and discarding of information respectively. By using these wild-cards, the language is endowed with the proper flexibility to generate complex search patterns with relatively simple syntax. However, the use of wild-card symbols suffers a serious implementation problem when applied to partially order sorted systems like Maude, because wild-cards must be later converted into variables whose sort (type) needs to be inferred.

A connected component (*kind* in Maude) is a set of sorts that are directly or indirectly related in the sub-sort ordering [CDE+11]. To properly infer the sort of each wild-card in Maude and any similar system, we must first obtain the information about all the connected components created by the partially ordered relation established in the declaration of sorts of the program whose trace is to be analyzed. Then, we must inspect each connected component and perform the necessary actions to ensure that all of them have exactly one top sort, namely by creating new sorts and establishing sub-sort relations for those connected components with more than one top sort. Finally, by creating one constant per wild-card and connected component top sort in the considered Maude program, the system parser is able to infer the proper sort by simple reading the pattern holding the wild-cards and matching our wild-card variables to the newly created program constants. Example 2.2.1 shows how *i*Julienne, modifies[1] a Maude program to properly infer the corresponding sorts of the proposed querying language wild-cards.

Example 2.2.1

The following Maude program defines a Maude specification for the minmax function.

```
mod MINMAX is
  inc INT .
  sorts NeList List Pair .
  subsorts Nat < List .

  op nil : -> List [ctor] .
```

[1]These operations are performed by *i*Julienne in a transparent manner to the user.

```
  op _;_ : List List -> List [ctor assoc] .
  op PAIR : Nat Nat -> Pair .
  op 1st : Pair -> Nat .
  op 2nd : Pair -> Nat .
  op Max : Nat Nat -> Nat .
  op Min : Nat Nat -> Nat .
  op minmax : List -> Pair .
  op Maxl : List -> Nat .

  var N X Y : Nat .
  var L : List .
  var P : Pair .

  crl [Max1] : Max(X,Y) => X if X >= Y .
  crl [Max2] : Max(X,Y) => Y if X < Y .

  crl [Min1] : Min(X,Y) => Y if X > Y .
  crl [Min2] : Min(X,Y) => X if X <= Y .

  rl [1st] : 1st(PAIR(X , Y )) => X .
  rl [2nd] : 2nd(PAIR(X , Y )) => Y .

  rl [minmax1] : minmax(N) =>  PAIR(N,N) .
  rl [minmax2] : minmax(N ; L) =>
                     PAIR(Min(N,1st(minmax(L))) , Max(N,2nd(minmax(L)))) .

  rl [MaxL] : Maxl(L) => 1st(minmax(L)) .

endm
```

The procedure that *i*Julienne runs in order to prepare the program for properly inferring the corresponding sorts is as follows.

First, we obtain the connected components for the `MINMAX` program in the following way:

```
  rewrite in iJulienne : getKinds(upModule('MINMAX, true)) .

  rewrites: 3 in 7635026678ms cpu (7ms real) (0 rewrites/second)
  result NeKindSet: ''[Bool‘] ; ''[NeList‘] ; ''[Pair‘] ; ''[List‘,Int‘]
```

We can observe that the partial order resulting from the declaration of sorts and subsorts in the `MINMAX` program identifies four connected components, three of them having only one top sort, namely `Bool`, `NeList` and `Pair`. However, the fourth connected component we retrieve has both `List` and `Int` as top sorts. In order to properly infer the type of a variable for this

connected component, *i*Julienne must create an artificial super-top and declare both `List` and `Int` sub-sorts of the recently created super-top.

This is easily achieved by adding the following declarations to the original `MINMAX` program:

```
sort JV2TOP0 .
subsorts List Int < JV2TOP0 .
```

The final step to achieve the automatic and transparent inferring of sorts is to create the set of constants that will match our wild-card variables by adding the following operators[2] to the `MINMAX` program:

```
op JV2VAR : -> Bool .
op JV2BOT : -> Bool .
op JV2VAR : -> JV2TOP0 .
op JV2BOT : -> JV2TOP0 .
op JV2VAR : -> NeList .
op JV2BOT : -> NeList .
op JV2VAR : -> Pair .
op JV2BOT : -> Pair .
```

At this point, our querying language syntax is fully integrated in the `MINMAX` program, allowing the user to perform a huge variety of queries to *easily* analyze any trace obtained by running the original program.

The versatility of our querying language is pictured in Example 2.2.2. In addition, Section 2.3 explains how to perform queries directly in the *i*Julienne tool.

Example 2.2.2

Consider the Maude specification of Example A together with the following initial program state:

```
< Alice | 50 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ;
credit(Alice,10) ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ;
transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ;
transfer(Bob,Charlie,4)
```

[2] JV2VAR and JV2BOT are two *i*Julienne reserved words that correspond to the `?` and `_` wild-cards respectively.

If we want to identify the amounts of money transferred by Alice to another user, we should query the state with the pattern `transfer(Alice,_,?)`, which discards the information related to the receiver and identifies the amounts transferred by Alice. After launching the query, we will identify that `Alice` has two transfer operations pending, with quantities `15` and `20`.

Consider now that we want to identify the names of the clients whose balance is 20. Then, we must launch a query using the pattern `< ? | 20 >`, which will identify `Bob`, `Charlie` and `Daisy` as the clients that match the given criterion.

Finally, consider that we want to search for the pattern `Charlie` in order to identify in how many operations is Charlie involved (including its own balance declaration). As a result, the term `Charlie` will be identified in the following operations: `< Charlie | 20 >`, `transfer(Alice,Charlie,15)`, `transfer(Bob,Charlie,4)` and `debit(Charlie,50)`.

At this point, one final source of eventual problems has to be addressed, namely ambiguity. Querying patterns may be as ambiguous as programs, specially when making use of wild-cards. In order to disambiguate patterns, the querying language implemented in *i*JULIENNE offers the possibility of either automatically infer the proper sort of each wild-card as explained previously, or let the user specify the sort in the same manner that dictates the syntax of Maude, that is, by concatenating wild-card and sort by means of the `:` character (e.g., `?:Nat`) or, in the case of a constant, by wrapping them in parentheses and concatenating the sort by means of a dot (e.g., `(cnt).Nat`). Example 2.2.3 shows how the disambiguation of a pattern provides widely different querying results.

Example 2.2.3

Consider the `MINMAX` program of Example 2.2.1 and the following state:

```
PAIR(Min(4,1st(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))
)))),Max(4,2nd(minmax(7 ; 0))))
```

Consider we want to identify those values appearing as single `minmax` parameters. Then, we may want to perform a query to search for the pattern `minmax(?)`. However, as `minmax` can hold either a single `Nat` or a `List` (of `Nat`) terms, we discover that the original pattern is somehow ambiguous, as it identifies two appearances of `minmax(0)` and one of `minmax(7 ; 0)`.

Nevertheless, if we take advantage of the disambiguation facilities that are provided in *i*JULIENNE, we may want to perform a query using the pattern `minmax(?:Nat)`, which will indeed recover only the two appearances of `minmax(0)` we wanted to identify, as they are the only terms that match the disambiguated query.

2.2.4 Program Slicing

Additionally to the backward trace slicing capabilities, *i*JULIENNE offers the possibility to perform a cogent form of *dynamic program slicing.*

A first, naïve approach to compute a dynamic program slice based on an execution trace would be to add to the program slice all and only those rules/equations that have been applied within the trace. After that, for every rule or equation in the slice, add the operators and variables appearing in them. Finally, for every operator and variable in the slice, we would add the corresponding sorts and sub-sorts. However, this approach is trivial and can be easily improved when we work with trace slices. The difference between a program slice based on execution traces and program a slice based on trace slices is the possibility to consider where the rule or equation was applied. In particular, only rules/equations applied in relevant positions have to be added to the program slice. Example 2.2.4 shows how *i*JULIENNE computes efficiently a program slice.

Example 2.2.4

Consider the rewrite theory of Example B together with the following trace and its corresponding trace slice.

<table>
<tr><th>Step</th><th>RuleName</th><th>Execution trace</th><th>Sliced trace</th></tr>
<tr><td>1</td><td>'Start</td><td>< Alice | 50 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Alice,10) ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>< Alice | * > ; * ; * ; * ; credit(Alice,*) ; * ; * ; * ; * ; * ; *</td></tr>
<tr><td>3</td><td>credit</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 60 ></td><td>* ; * ; * ; * ; * ; * ; * ; * ; * ; < Alice | * ></td></tr>
<tr><td>6</td><td>credit</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 60 ></td><td>< Alice | * > ; * ; * ; * ; * ; * ; * ; * ; *</td></tr>
<tr><td>9</td><td>debit</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | - 30 ></td><td>< Alice | * > ; * ; * ; * ; * ; * ; * ; *</td></tr>
<tr><td>12</td><td>debit</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 55 ></td><td>< Alice | * > ; * ; * ; * ; * ; * ; *</td></tr>
</table>

```
mod BANK_ERR is inc INT .
sorts Account Msg State Id .
subsorts Account Msg < State .
var Id Id1 Id2 : Id .
var bal bal1 bal2 newBal newBal1 newBal2 M : Int .
op empty-state : -> State .
op Alice : -> Id .
op Bob : -> Id .
op Charlie : -> Id .
op Daisy : -> Id .
op _;_ : State State -> State [assoc comm] .
op <_|_> : Id Int -> Account [ctor] .
ops credit debit : Id Int -> Msg [ctor] .
op transfer : Id Id Int -> Msg [ctor] .

crl [credit] : credit(Id, M) ; < Id | bal > => < Id | newBal >
if newBal := bal + M .

crl [debit] : debit(Id, M) ; < Id | bal > => < Id | newBal >
if newBal := bal - M .

crl [transfer] : transfer(Id1, Id2, M) ; < Id1 | bal1 > ; < Id2 | bal2 > => < Id1 | newBal1 > ; < Id2 | newBal2 >
if debit(Id1, M) ; < Id1 | bal1 > => < Id1 | newBal1 >
/\ credit(Id2, M) ; < Id2 | bal2 > => < Id2 | newBal2 > .

endm
```

Figure 2.4: Dynamic program slice of Example 2.2.4.

The naïve program slicing methodology sketched above would point as relevant all the rules/equations[3] applied along the trace (i.e., `credit` in steps 3 and 6 and `debit` in steps 9 and 12), and thus compute the corresponding program slice by calculating all the dependence previously explained.

In contrast, our dynamic program slicing based on trace slices considers as relevant only the information derived from those rules/equations applied at the relevant positions of our trace slice (i.e., `credit` in step 3), and therefore safely ignores the `debit` rule as well as the application of rule `credit` in step 6, which would point Daisy as relevant if taken into account, whereas in no way it is related to the relevant information of our trace slice. The resulting program slice is depicted in Figure 2.4.

2.3 *i*JULIENNE at work

To illustrate how *i*JULIENNE works in practice, we show two typical analysis sessions that illustrate how *i*JULIENNE works in practice. In the first session, a simple planning system is debugged by intensively exploiting trace and program slicing, while the second session highlights *i*JULIENNE analysis

[3]Flattening and unflattening transformation are not considered for this task.

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

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

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

Figure 2.5: `BLOCKS-WORLD` faulty Maude specification.

capabilities on a real-size Maude specification that encodes a Webmail application. The considered examples (together with several others) are available at [iJu12] and the described analysis and debugging sessions can be reproduced by accessing *i*JULIENNE through its online Web interface.

2.3.1 Debugging the Blocks World Example

Blocks World is one of the most famous planning problems in artificial intelligence. We assume that there are some blocks, placed on a table, that can be moved by means of a robot arm; the goal of the robot arm is to produce one or more vertical stacks of blocks. In our specification, which is shown in the Maude module `BLOCKS-WORLD` of Figure 2.5, we define a Blocks World system with three different kinds of blocks that are defined by means of the operators `a`, `b`, and `c` of sort `Block`. Different blocks have different sizes that are described by using the unary operator `size`. We also consider some operators that formalize block and robot arm properties whose intuitive meanings are given in the accompanying program comments.

The states of the system are modeled by means of associative and commutative lists of properties of the form $\texttt{prop}_1\&\texttt{prop}_2\&\ldots\&\texttt{prop}_n$, which

describe any possible configuration of the blocks on the table as well as the status of the robot arm.

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

Barely perceptible, the Maude specification of Figure 2.5 fails to provide a correct Blocks World implementation. By using the `BLOCKS-WORLD` module, it is indeed possible to derive system states that represent erroneous configurations. For instance, the initial state

$\mathtt{s}_\mathtt{i}$ = `empty` & `clear`(`a`) & `table`(`a`) & `clear`(`b`) & `table`(`b`) & `clear`(`c`) & `table`(`c`)

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

$\mathtt{s}_\mathtt{f}$ = `empty` & `empty` & `table`(`b`) & `table`(`c`) & `clear`(`a`) & `clear`(`c`) & `on(a,b)`

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

To find the cause of this wrong behavior, we feed *i*JULIENNE with the faulty rewrite sequence $\mathcal{T} = \mathtt{s}_\mathtt{i} \rightarrow^* \mathtt{s}_\mathtt{f}$, and we initially slice $\mathcal{T}$ w.r.t. the slicing criterion that observes the two anomalous occurrences of the `empty` property and the stack `on`(`a`, `b`) in State $\mathtt{s}_\mathtt{f}$. This task can be easily performed in *i*JULIENNE by first highlighting the terms that we want to observe in State $\mathtt{s}_\mathtt{f}$ with the mouse pointer and then starting the slicing process. *i*JULIENNE yields a trace slice that simplifies the original trace by recording only those data that are strictly needed to produce the considered slicing criterion. Also, it automatically computes the corresponding program slice, which consists of the equations defining the `size` operator together with the `pickup` and `stack` rules (see Figure 2.6). This allows us to deduce that the malfunction is located in one or more rules and equations that are included in the program slice.

The generated trace slice is then browsed backwards using the *i*JULIENNE's navigation facility in search of a possible explanation for the wrong behav-

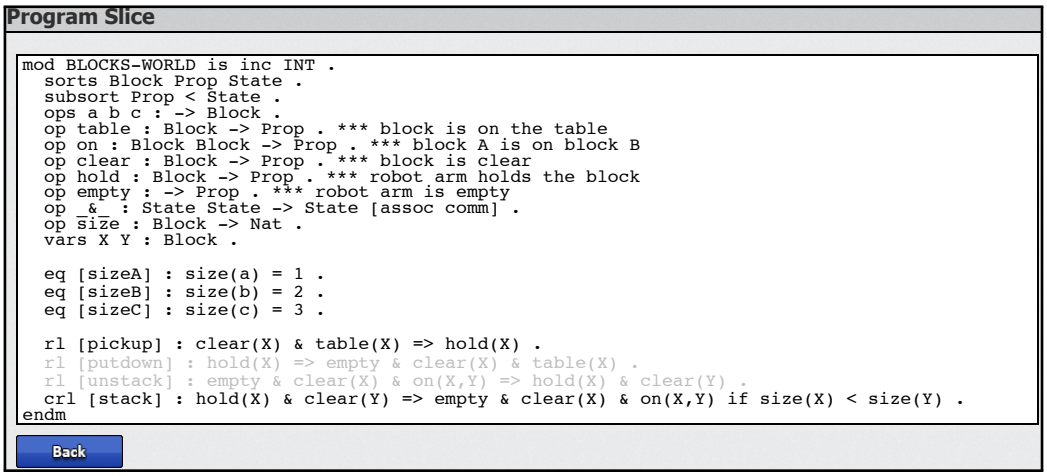

Figure 2.6: Program slice computed w.r.t. the slicing criterion `empty`, `empty`, `on(a, b)`.

ior. During this phase, we focus our attention on State 3 (Figure 2.7), which is inconsistent since it models a robot arm that is holding block `a` and is empty at the same time. Therefore, we decide to further refine the trace slice by incrementally applying backward trace slicing to State 3 w.r.t. the slicing criterion `hold(a)`. This way we achieve a supplementary reduction of the previous trace slice in which we can easily observe that `hold(a)` only depends on the `clear(a)` and `table(a)` properties (see Figure 2.8). Furthermore, the computed program slice includes the single `pickup` rule (see Figure 2.9). Thus, we can conclude that:

1. the malfunction is certainly located in the `pickup` rule (since the computed program slice only contains that rule);

2. the `pickup` rule does not depend on the status of the robot arm (this is witnessed by the fact that `hold(a)` only relies on the `clear(a)` and `table(a)` properties);

3. by 1 and 2, we can deduce that the `pickup` rule is incorrect, as it never checks the emptiness of the robot arm before grasping a block.

A possible fix of the detected error consists in including the `empty` property in the left-hand side of the `pickup` rule, which enforces the robot arm

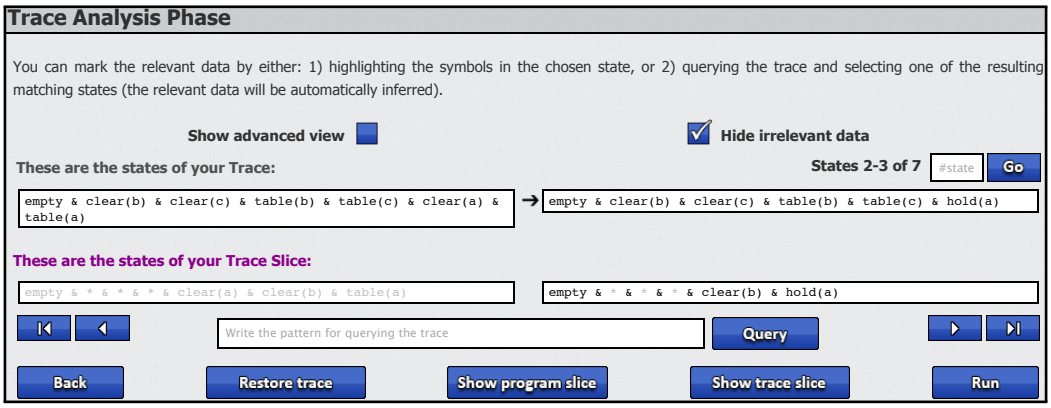

Figure 2.7: Navigation through the trace slice of the *Blocks World* example.

to always be idle before picking up a block. The corrected version of the rule is hence as follows:

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

2.3.2 Analyzing a Webmail Application

In this section, we reproduce a typical trace analysis session with *i*JULIENNE that operates on a Maude specification of a realistic Webmail application that consists of 10 rewrite rules and 134 equations. The specification models both server-side aspects (e.g., Web script evaluations, database interactions) and browser-side features (e.g., forward/backward navigation, Web page refreshing, window/tab openings). The Web application behavior is formalized by using rewrite rules of the form $[\mathsf{label}] : \mathsf{WebState} \Rightarrow \mathsf{WebState}$, where WebState is a triple that we represent with the following operator $\_\|\_\|\_ : (\mathsf{Browsers} \times \mathsf{Message} \times \mathsf{Server}) \rightarrow \mathsf{WebState}$ that can be interpreted as a system shot that captures the current configuration of the active browsers (i.e., the browsers currently using the Webmail application) together with the channel through which the browsers and the server communicate via message-passing. An execution trace specifies a sequence of `WebState` transitions that represents a possible execution of the Webmail application.

The session starts by loading the Maude Webmail specification, together with the execution trace to be analyzed, into the *i*JULIENNE trace analyzer.

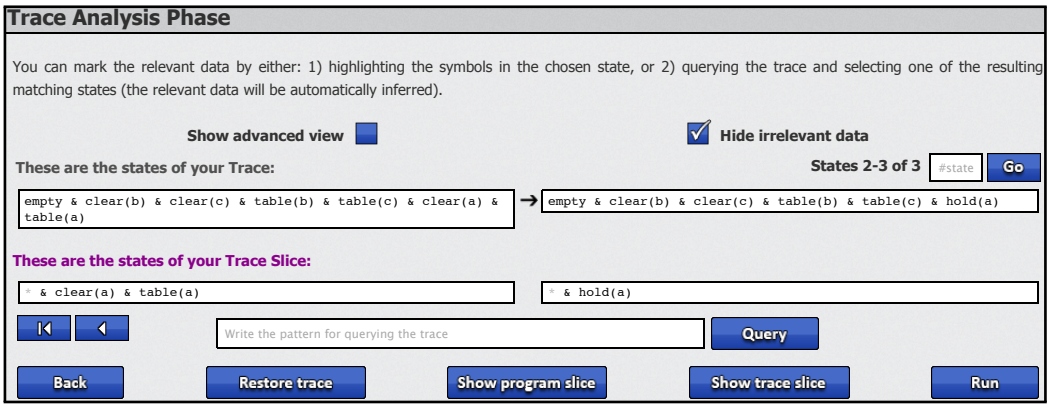

Figure 2.8: Navigation though the refined trace slice of the *Blocks World* example.

The trace can be directly pasted in the input form or uploaded from a trace file that was written off-line. It can also be dynamically computed by the system (using Maude meta-search capabilities) by introducing the initial and final states of the trace. In Figure 2.10, we directly fed *i*Julienne with an execution trace that represents a counter-example that was automatically generated by the Maude LTLR model-checker. The considered trace consists of 97 states, each of which has about 5.000 characters.

The aim of our analysis is to extract the navigation path of a possible malicious user `idA` within the Web application from the execution trace. This is particularly hard to perform by hand since the trace is extremely large and system states contain a huge amount of data.

Therefore, we decide to slice the trace with *i*Julienne in order to isolate only the Web interactions related to user `idA`, getting rid of all the remaining unrelated information. To this end, we define the query `B(idA, _, ?, _, _, _, _, _, _)`, which allows us to select all states in the trace that contain a browser data structure that is associated with user `idA`. Note that only the third argument of the browser data structure, which corresponds to the Web page displayed on the browser, is declared relevant in the query (i.e., it is marked by the card ?). Indeed, we are interested in tracking only the Web pages visited by the user `idA`. The remaining information (such as script evaluation, and Web interactions with other users) is not pinpointed and, hence, will be systematically removed by the slicing tool (and replaced by the symbol $*$) which

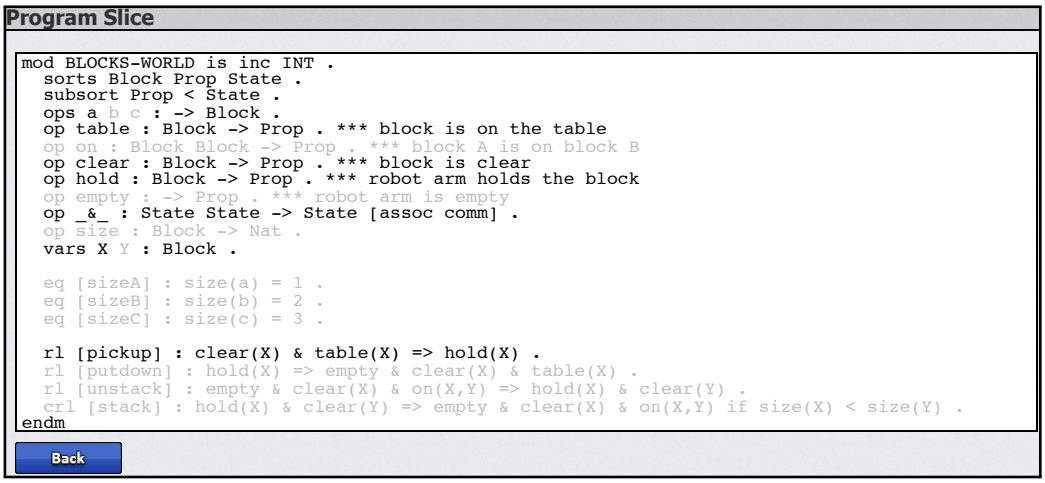

Figure 2.9: Program slice computed w.r.t. the slicing criterion `hold(a)`.

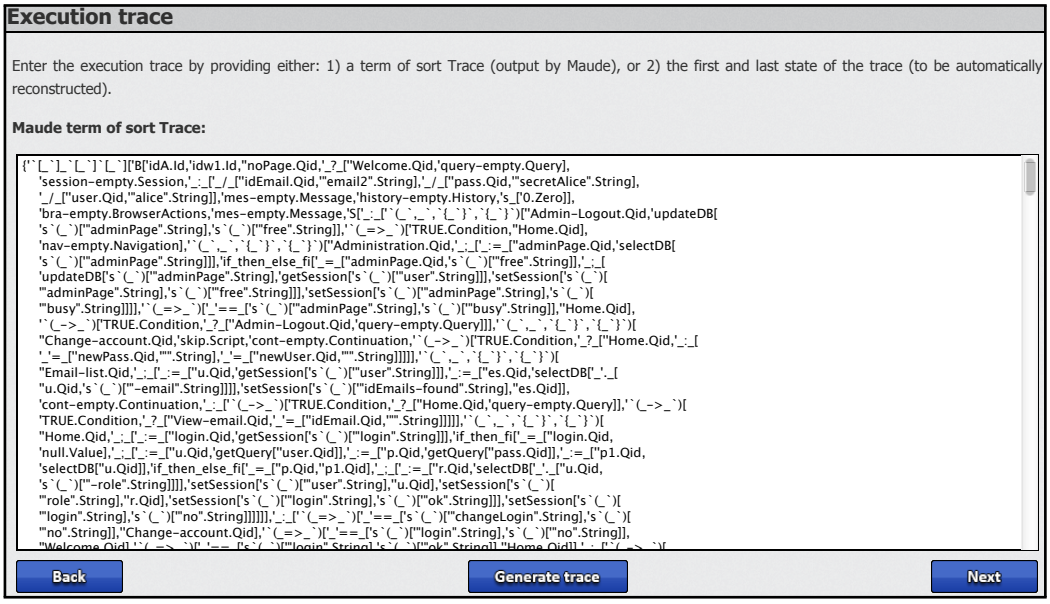

Figure 2.10: Loading the Webmail execution trace.

facilitates comprehension. By running the query, *i*JULIENNE computes the outcome shown in Figure 2.11, which delivers the states that satisfy the

query. More concretely, the first 20 states of the trace do match the query, while the remaining 77 states do not include any Web interaction with the user `idA` and thus do not need to be inspected.

Now, since we are interested in observing the whole navigation history of user `idA`, we select the last state in the trace that matches the query, namely State 20, and we apply backward trace slicing on that state in order to generate a simplified view of the first 20 states of the trace. Note that the slicing criterion is automatically computed by *i*Julienne by extracting the data matching the query from State 20 (specifically, the target symbols `B`, `idA`, and the current browser Web page `Welcome`).

After pressing the *Run* button, we get a browsable trace slice (see Figure 2.12) where each state of the trace slice is purged of the irrelevant data w.r.t. the slicing criterion, and all the rewrite steps that do not affect the observed data are marked as irrelevant and are simply removed from the slice, which further reduces its size. The reduction rate achieved w.r.t. the original trace reaches an impressive 91%; Specially, 88 of the 97 states were found to be redundant with regard to the selected slicing criterion. This makes the trace slice easy to analyze by hand. Actually, by navigating through the trace slice, it can be immediately observed that the malicious user `idA` visits the `Login` page, succeeds to log onto the Webmail system and enters the Webmail `Welcome` page.

2.4 Experimental Evaluation

In order to properly assess the maturity and effectiveness of *i*Julienne, we have carried out some experiments by testing *i*Julienne on rather large execution traces, such as the counter-examples delivered by the Maude LTLR model-checker [CDH+07] and several other case studies that are available at the *i*Julienne Web site [iJu12]. More precisely, we have considered:

- two runs of a fault-tolerant client-server communication protocol (`FTCP`) specified in Maude that aim at extracting information related to a specific server and client in a scenario that involves multiple servers and clients.

- two execution traces generated by Maude-NPA [EMM09], which is an RWL-based analysis tool for cryptographic protocols that takes into

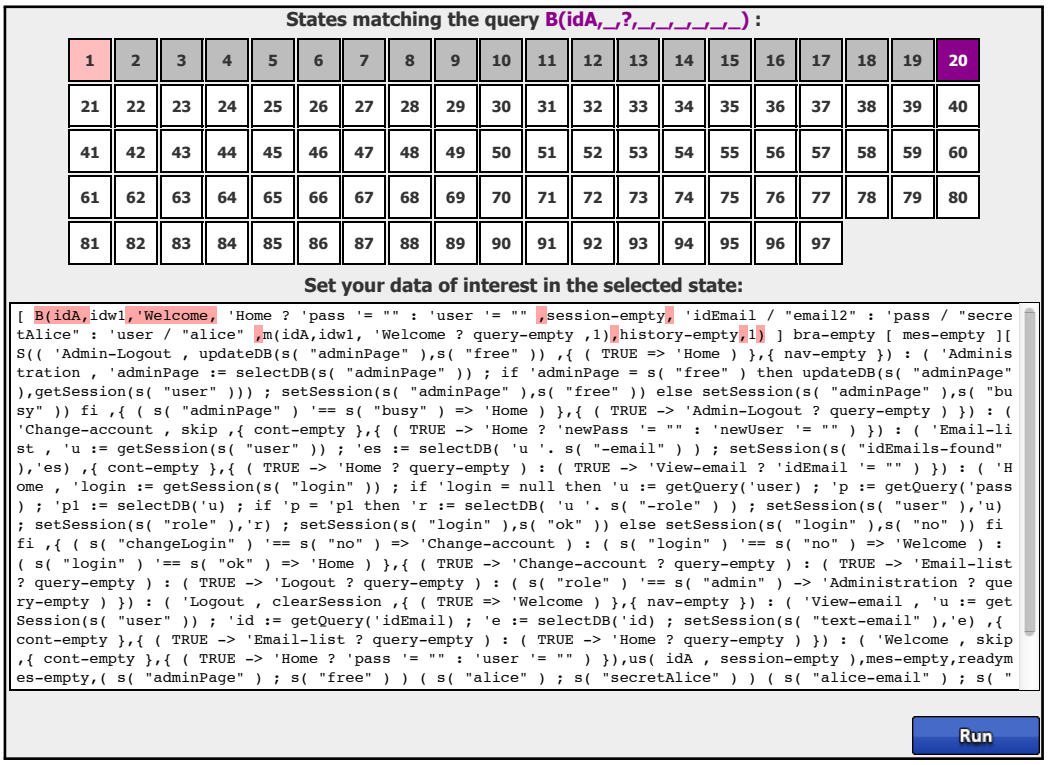

Figure 2.11: Querying the Webmail trace w.r.t. B(idA, _, ?, _, _, _, _, _, _)

account many of the algebraic properties of cryptosystems. These include the cancellation of encryption and decryption, Abelian groups (including exclusive-or), exponentiation, and homomorphic encryption. The considered traces model two instances of a well-known, man-in-the-middle attack to the Needham-Schroeder network authentication protocol where we select the intruder's actions and knowledge while discarding all the remaining information in the protocol session.

- two counter-examples produced by model-checking a real-size Webmail application specified in Web-TLR [ABER11]. The slicing criteria allowed several critical data to be isolated and inspected (e.g., the navigation of a malicious user), the messages exchanged by a specific Web browser with the Webmail server, and session data of interest (e.g., cookies).

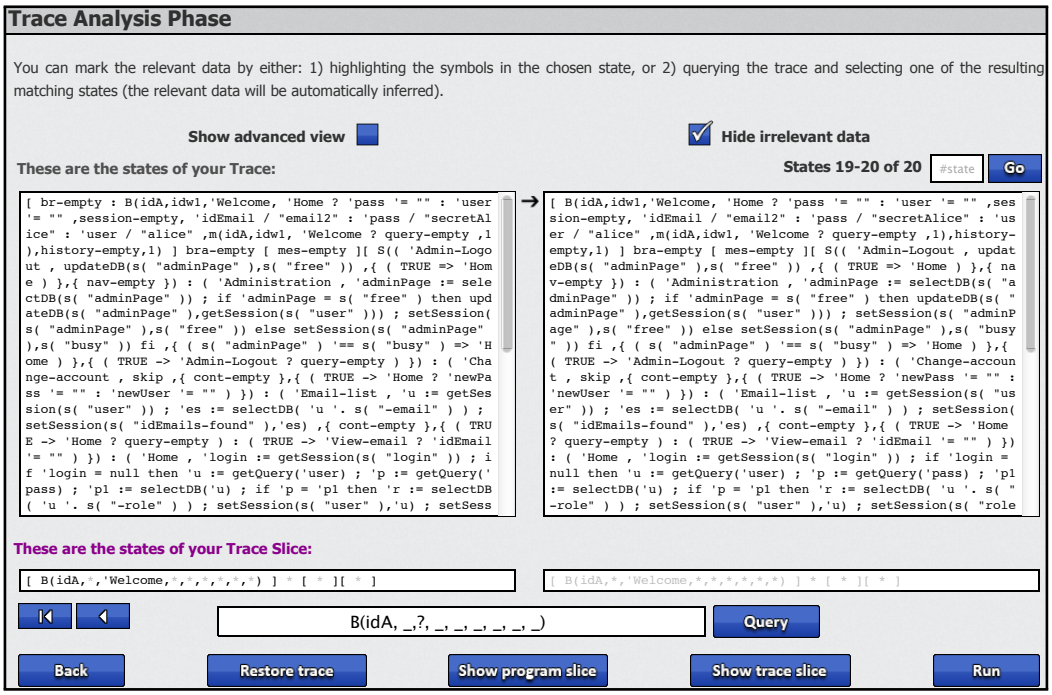

Figure 2.12: Webmail trace slice after querying the trace

Example trace	Original trace size	Slicing criterion	Sliced trace size	% reduction	% reduction by changing criterion
FTCP.$\mathcal{T}_1$	2054	FTCP.$\mathcal{T}_1$.O_1	294	85.69%	97.89%
		FTCP.$\mathcal{T}_1$.O_2	316	84.62%	97.30%
FTCP.$\mathcal{T}_2$	1286	FTCP.$\mathcal{T}_2$.O_1	135	89.40%	98.11%
		FTCP.$\mathcal{T}_2$.O_2	97	92.46%	99.01%
Maude-NPA.$\mathcal{T}_1$	21265	Maude-NPA.$\mathcal{T}_1$.O_1	2249	89.42%	98.12%
		Maude-NPA.$\mathcal{T}_1$.O_2	2261	89.36%	98.03%
Maude-NPA.$\mathcal{T}_2$	34681	Maude-NPA.$\mathcal{T}_2$.O_1	3015	91.30%	99.08%
		Maude-NPA.$\mathcal{T}_2$.O_2	3192	90.79%	98.84%
Web-TLR.$\mathcal{T}_1$	38983	Web-TLR.$\mathcal{T}_1$.O_1	2045	94.75%	99.28%
		Web-TLR.$\mathcal{T}_1$.O_2	2778	92.87%	99.14%
Web-TLR.$\mathcal{T}_2$	69491	Web-TLR.$\mathcal{T}_2$.O_1	8493	87.78%	97.99%
		Web-TLR.$\mathcal{T}_2$.O_2	5034	92.76%	99.05%
		% reduction average		90.10%	98.49%

Table 2.1: Incremental slicing benchmarks.

The results of our experiments are shown in Table 2.1. The execution traces for the considered cases consist of sequences of 10–1000 states, each of which contains from 60 to 5000 characters. In all the experiments, not only do the trace slices that we obtained show impressive reduction rates (ranging from ∼79% to ∼98%), but we were also even able to strikingly improve these rates by an average of 8.5% (ranging from ∼97% to ∼99%) by using *incremental slicing*. In most cases, the delivered trace slices were cleaned enough to be easily analyzed, and we noted an increase in the effectiveness of the analysis processes. With regard to the scalability and time required to perform the analyses, *i*JULIENNE is extremely time efficient; the elapsed times are small even for very complex traces and scale linearly. For example, running the analyzer for a 20Kb trace w.r.t. a Maude specification with about 150 rules and equations (with AC rewrites) took less than 1 second (480.000 rewrites per second on standard hardware, 2.26GHz Intel Core 2 Duo with 4Gb of RAM memory).

Conclusions and future work

We have developed a new, incremental backward trace slicing methodology [ABFS13b] by extending the backward trace slicing technique for conditional rewriting logic specifications described in [ABFR12a]. Our methodology can highly improve the already impressive reduction rates achieved by the original technique while preserving its demonstrated efficiency when dealing with huge and complex execution traces such as those generated by the Maude LTL model checker [EMS02, EMS03].

We have implemented our methodology in *i*Julienne [iJu12], which is the first slicing-based, incremental trace analysis tool for rewriting logic that greatly reduces the size of the computation traces and can make their analysis feasible even for complex, real-size applications. Our tool conveys an incremental slicing approach where the user can refine the slicing criteria and then the extra irrelevant contents (i.e., the differences between the slices) are automatically done away with. *i*Julienne is also endowed with useful debugging features such as trace querying and program slicing, which serve as a suitable tandem for the slicing-based analysis task. In addition, it is important to note that our trace analyzer does not remove any information that is relevant, independently of the skills of the user.

Our ongoing research on trace slicing for rewriting logic computations focuses on exploring the forward approach in the new tool Anima [Ani13], an online stepper for Maude programs that implements a novel, parametrized technique for the inspection of rewriting logic computations described in [ABFS13a].

Finally, as future work, we plan to explore other challenging applications of our trace slicing methodology, such as runtime verification [BFF+10] which is concerned with monitoring and analysis of system executions. Consider a programming language $\mathcal{L}$ which is given a RWL executable semantics. Then one can use the semantics as an interpreter to execute $\mathcal{L}$ programs (given as terms) directly within the semantics of their programming language as in [FCMR04], and hence Maude can be used to trace such executions. Then, by querying the trace slice w.r.t. a reference specification, runtime verification might be semantically grounded in our setting while it is commonly off-hacked in more traditional approaches by means of program instrumentation.

Bibliography

[ABE+11] M. Alpuente, D. Ballis, J. Espert, F. Frechina, and D. Romero. Debugging of Web Applications with WEB-TLR. In *Proc. of 7th International Workshop on Automated Specification and Verification of Web Systems (WWV 2011)*, volume 61 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 66–80. Open Publishing Association, 2011.

[ABER10] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Model-checking Web Applications with Web-TLR. In *Proc. of 8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *Lecture Notes in Computer Science (LNCS)*, pages 341–346. Springer-Verlag, 2010.

[ABER11] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward Trace Slicing for Rewriting Logic Theories. In *Proc. of 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *Lecture Notes in Computer Science (LNCS)*, pages 34–48. Springer-Verlag, 2011.

[ABFR12a] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Backward Trace Slicing for Conditional Rewrite Theories. In *Proc. of the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2012)*, volume 7180

of *Lecture Notes in Computer Science (LNCS)*, pages 62–76. Springer-Verlag, 2012.

[ABFR12b] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Julienne: A Trace Slicer for Conditional Rewrite Theories. In *Proc. of the 18th International Symposium on Formal Methods (FM 2012)*, volume 7436 of *Lecture Notes in Computer Science (LNCS)*, pages 28–32. Springer-Verlag, 2012.

[ABFR12c] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Using Conditional Trace Slicing for Improving Maude Programs. *Science of Computer Programming*, 2012. Under second revision.

[ABFS13a] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Parametric Exploration of Rewriting Logic Computations. In *Proc. of the 5th International Symposium on Symbolic Computation in Software Science (SCSS 2013)*, volume 15 of *EasyChair Proceedings in Computing (EPiC)*, pages 4–18. EasyChair, 2013.

[ABFS13b] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Slicing-Based Trace Analysis of Rewriting Logic Specifications with iJulienne. In *Proc. of the 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science (LNCS)*, pages 121–124. Springer-Verlag, 2013.

[ABR09] M. Alpuente, D. Ballis, and D. Romero. Specification and Verification of Web Applications in Rewriting Logic. In *Proc. of the 16th International Symposium on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science (LNCS)*, pages 790–805. Springer-Verlag, 2009.

[ADS93] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software Practice and Experience*, 23(6):589–616, 1993.

[Agr91] H. Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, West Lafayette, IN 47907, September 1991. Ph. D. Thesis.

[Ani13] The ANIMA web site, 2013. Available at: `http://safe-tools.dsic.upv.es/anima/`.

[BBF09] M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In *Proc. of the 7th International Conference on Computational Methods in Systems Biology (CMSB 2009)*, volume 5688 of *Lecture Notes in Computer Science (LNCS)*, pages 68–82. Springer-Verlag, 2009.

[BFF+10] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *Proc. of the 1st International Conference on Runtime Verification (RV 2010)*, volume 6418 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2010.

[BKdV00] I. Bethke, J. W. Klop, and R. de Vrijer. Descendants and Origins in Term Rewriting. *Information and Computation*, 159(1–2):59–124, 2000.

[BKdV03] M. Bezem, J. W. Klop, and R. de Vrijer. *Term Rewriting Systems (TeReSe)*. Cambridge University Press, Cambridge, UK, 2003.

[BM06] R. Bruni and J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science (TCS)*, 360(1–3):386–414, 2006.

[BM12] K. Bae and J. Meseguer. A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. In *Proc. of the 9th International Workshop on Rule-Based Programming (RULE 2008)*, volume 290 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 19–36. Elsevier Science Publishers Ltd., 2012.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science (TCS)*, 285(2):187–243, 2002.

[CDE+11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.6). Technical report, SRI International Computer Science Laboratory, 2011. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.

[CDH+07] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. C. Ölveczky. The Maude Formal Tool Environment. In *Proc. of the 2nd Algebra and Coalgebra in Computer Science (CALCO 2007)*, volume 4624 of *Lecture Notes in Computer Science (LNCS)*, pages 173–178. Springer-Verlag, 2007.

[CR09] F. Chen and G. Rosu. Parametric Trace Slicing and Monitoring. In *Proc. of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *Lecture Notes in Computer Science (LNCS)*, pages 246–261. Springer-Verlag, 2009.

[CRW00] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *Lecture Notes in Computer Science (LNCS)*, pages 176–193. Springer-Verlag, 2000.

[DM10] F. Durán and J. Meseguer. A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In *Proc. of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA 2010)*, volume 6381 of *Lecture Notes in Computer Science (LNCS)*, pages 86–103. Springer-Verlag, 2010.

[EMM09] S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V (FOSAD 2007/2008/2009 Tutorial Lectures)*, volume 5705 of *Lecture Notes in Computer Science (LNCS)*, pages 1–50. Springer-Verlag, 2009.

[EMS02] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In *Proc. of the 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 162–187. Elsevier Science Publishers Ltd., 2002.

[EMS03] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and Its Implementation. In *Proc. of the 10th International SPIN Workshop (SPIN 2003)*, volume 2648 of *Lecture Notes in Computer Science (LNCS)*, pages 230–234. Springer-Verlag, 2003.

[FCMR04] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal Analysis of Java Programs in JavaFAN. In *Proc. of the 16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science (LNCS)*, pages 501–505. Springer-Verlag, 2004.

[FT94] J. Field and F. Tip. Dynamic Dependence in Term rewriting Systems and its Application to Program Slicing. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP 1994)*, volume 844 of *Lecture Notes in Computer Science (LNCS)*, pages 415–431. Springer-Verlag, 1994.

[HCS+96] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. The Mercury Language Reference Manual. Technical report, University Of Melbourne, 1996.

[HHJW07] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proc. of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55. ACM, 2007.

[HKF95] T. Hoffner, M. Kamkar, and P. Fritzson. Evaluation of Program Slicing Tools. In *AADEBUG*, pages 51–69, 1995.

[iJu12] The *i*JULIENNE web site, 2012. Available at: `http://safe-tools.dsic.upv.es/iJulienne/`.

[KL88] B. Korel and J. Laski. Dynamic Program Slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[Klo92] 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.

[Mac05] I. D. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, University of Melbourne, 2005. Ph. D. Thesis.

[Mes90a] J. Meseguer. A Logical Theory of Concurrent Objects. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP 1990)*, pages 101–115. ACM, 1990.

[Mes90b] J. Meseguer. Conditional Rewriting Logic: Deduction, Models and Concurrency. In *Proc. of the 2nd International Conditional and Typed Rewriting Systems Workshop (CTRS 1990)*, volume 516 of *Lecture Notes in Computer Science (LNCS)*, pages 64–91. Springer-Verlag, 1990.

[Mes91] J. Meseguer. Rewriting as a Unified Model of Concurrency. *SIGPLAN OOPS Messenger*, 2(2):86–88, 1991.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science (TCS)*, 96(1):73–155, 1992.

[Plo04] G. D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[RB06] N. F. Rodrigues and L. Soares Barbosa. Component Identification Through Program Slicing. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 160:291–304, 2006.

[Wei79] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979. AAI8007856.

[Wei81] M. Weiser. Program Slicing. In *Proc. of the 5th International Conference on Software (ICSE 1981)*, pages 439–449. IEEE Computer Society Press, 1981.

[Wei82] M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wil85] M. Wilkes. *Memoirs of a Computer Pioneer*. MIT Press Series in the History of Computing. Cambridge, MA: The MIT Press, 1985.