UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

A Rewriting-based, Parameterized Exploration Scheme for the Dynamic Analysis of Complex Software Systems

Ph.D. Thesis

Presented by:
Francisco Frechina

Supervisors:
María Alpuente
Demis Ballis

Valencia, October 2014

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS
INFORMÁTICOS Y COMPUTACIÓN

A Rewriting-based, Parameterized Exploration Scheme for the Dynamic Analysis of Complex Software Systems

Ph.D. Thesis

A dissertation submitted by *Francisco Frechina* in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science with International Mention at the Universitat Politècnica de València.

Valencia, October 2014

Presented by

Francisco Frechina Navarro

Supervisors

María Alpuente Frasnedo	Demis Ballis
Universitat Politècnica de València	Università di Udine

External evaluators

Marco Comini	Università di Udine
Francisco Durán	Universidad de Málaga
Luigi Liquori	INRIA Sophia Antipolis Méditerranée

Jury

Isidro Ramos	Universitat Politècnica de València	(President)
Rafael Caballero	Universidad Complutense de Madrid	(Secretary)
Luigi Liquori	INRIA Sophia Antipolis Méditerranée	(Vocal)

This work has been partially supported by the Spanish MEC project ref. TIN2010-21062-C02-02, by Generalitat Valenciana ref. PROMETEO2011/052, and by FPU-ME grant AP2010-5681.

If you live each day as if it was your last, someday you'll most certainly be right.

Si vives cada día como si fuera el último, algún día estarás en lo cierto.

Steve Jobs

Acknowledgements

Many years ago an old high school teacher asked me a question that made me think: Why do what someone else has done before? Since then, before doing something, I always first search on Google to see if I can take advantage, even partially, of what someone else has done before in order to avoid redoing it. To do otherwise would be a waste of time.

This led me (as could not be otherwise) to google '*thesis acknowledgments*'. I read several paragraphs about giving thanks, and I soon realized that each one of them had a great personal touch. I kept looking for inspiration and did not give up until I had found a tutorial that said —*writing thesis acknowledgments is as easy as remembering all of the sacrifices that you have made during your thesis time*—. This is the point where I stopped looking. Sacrifices? Hardly! I am so privileged for having been surrounded by exciting people who have made my time as a Ph.D. student one of the best times of my life.

The entrepreneurial side of my personality prevents me from doing what the majority of people do, which is why this section of my thesis is really complicated for me. Some people decide to write a long list of names, while others either declare their love to someone or thank God. The truth is that if I had to make a list of all of the people I should thank for their influence on my life throughout my scientific career, I am sure I would forget more than one (not because the list is very extensive, but rather because sometimes I am absent-minded about these things). But in all honesty, I am sure that none of those people will need to see their names here to know that they have been very important to me and that I will always be deeply grateful for sharing all the great moments we have had in recent years. They all catapulted my enthusiasm during my career, and even though my timidity does not allow me to make a declaration of love, I must admit that the appearance of the love of my life, the support of my family and my great friends have all helped me to maintain the enthusiasm to complete my doctoral thesis.

Though common as it may be, I feel impelled to dedicate some very special words to María Alpuente and Demis Ballis. As my thesis ad-

visors, they have oriented, inspired, and supported me in my scientific endeavors with an attention and devotion that have far exceeded any of my expectations. Being able to attend scientific conferences during my Ph.D. period allowed me to meet researchers from around the world, many of whom complained about lack of attention from their advisors. In this regard, I have been very lucky. I can never thank María and Demis enough for their continued and dedicated corrections and revisions of my work all of these years. Furthermore, I must not forget to mention my deep appreciation for the other co-authors of my latest works, Daniel Romero and Julia Sapiña, without whose collaboration we would never have achieved the level of quality that we have definitely reached. I must also give honorable mention to all of my colleagues from ELP, especially each of my (former) lab partners. Even though we still have not gotten a whiteboard, we do have a great sense of humor that has given us so many unforgettable moments.

I cannot conclude this part of my thesis without paraphrasing one of the people who has most inspired me throughout my life.

— Do you want to sell smoke for the rest of your life? Or do you want to change the world?

Fran
Valencia, October 2014

Abstract

Today's software systems are complex artifacts whose behavior is often extremely difficult to understand. This fact has led to the development of sophisticated formal methodologies for program analysis, comprehension, and debugging.

Trace analysis is concerned with techniques that allow execution traces to be dynamically searched for specific contents. The search can be carried out *forward* or *backward*. While forward analysis results in a form of *impact analysis* that identifies the scope and potential consequences of changing the program input, backward analysis allows *provenance analysis* to be performed; i.e., it shows how (parts of) a program output depends on (parts of) its input and helps estimate which input data need to be modified to accomplish a change in the outcome.

In this thesis, we investigate a number of trace analysis methodologies that are suitable for analyzing complex, textually-large execution traces in rewriting logic (RWL), which is a *logical* and *semantic framework* particularly suitable for formalizing highly concurrent, complex systems.

The first part of the thesis is devoted to develop an incremental, slicing-based backward trace analysis technique that achieves huge reductions in the size of the trace. This methodology favors better analysis and debugging since most tedious and irrelevant inspections that are routinely performed during diagnosis and bug localization can be eliminated automatically. This technique is illustrated by means of several examples that we execute by using the *i*Julienne system, an interactive trace slicer that we developed which implements the backward trace analysis technique.

The second part of the thesis formalizes a rich and highly dynamic, parameterized scheme for exploring rewriting logic computations. The scheme implements a generic animation algorithm that allows the nondeterministic execution of a given *conditional* rewrite theory to be followed up by using different modalities, including *incremental stepping* and *automated forward/backward slicing*, which drastically reduce the size and complexity of the traces under examination and allow users to evaluate

the effects of a given statement or instruction in isolation, track input change impact, and gain insight into program behavior (or misbehavior). Moreover, cutting down the execution trace can expose opportunities for program optimizations. With this methodology, an analyst can browse, slice, filter, or search the traces as they come to life during the program execution. The generic trace analysis framework has been implemented into the Anima system and we report a thorough experimental evaluation that we conducted which assesses the usefulness of the proposed approach.

Resumen

Los sistemas software actuales son artefactos complejos cuyo comportamiento es a menudo extremadamente difícil de entender. Este hecho ha llevado al desarrollo de metodologías formales muy sofisticadas para el análisis, comprensión y depuración de programas.

El análisis de trazas de ejecución consiste en la búsqueda dinámica de contenidos específicos dentro de las trazas de ejecución de un cierto programa. La búsqueda puede llevarse a cabo hacia adelante o hacia atrás. Si bien el análisis hacia adelante se traduce en una forma de análisis de impacto que identifica el alcance y las posibles consecuencias de los cambios en la entrada del programa, el análisis hacia atrás permite llevar a cabo un rastreo de la procedencia; es decir, muestra cómo (partes de) la salida del programa depende de (partes de) su entrada y ayuda a estimar qué dato de la entrada es necesario modificar para llevar a cabo un cambio en el resultado.

En esta tesis se investiga una serie de metodologías de análisis de trazas que son especialmente adecuadas para el análisis de trazas de ejecución largas y complejas en la *lógica de reescritura*, que es un marco lógico y semántico especialmente adecuado para la formalización de sistemas altamente concurrentes.

La primera parte de la tesis se centra en desarrollar una técnica de análisis de trazas hacia atrás que alcanza enormes reducciones en el tamaño de la traza. Esta metodología se basa en la fragmentación (*slicing*) incremental y favorece un mejor análisis y depuración ya que la mayoría de las inspecciones, tediosas e irrelevantes, que se realizan rutinariamente en el diagnóstico y la localización de errores se pueden eliminar de forma automática. Esta técnica se ilustra por medio de varios ejemplos que ejecutamos mediante el sistema iJulienne, una herramienta interactiva de fragmentación que hemos desarrollado y que implementa la técnica de análisis de trazas hacia atrás.

En la segunda parte de la tesis se formaliza un sistema paramétrico, flexible y dinámico, para la exploración de computaciones en la lógica de reescritura. El esquema implementa un algoritmo de animación genérico

que permite la ejecución indeterminista de una teoría de reescritura condicional dada y que puede ser objeto de seguimiento mediante el uso de diferentes modalidades, incluyendo una ejecución gradual paso a paso y una fragmentación automática hacia adelante (y/o hacia atrás), lo que reduce drásticamente el tamaño y la complejidad de las trazas bajo inspección y permite a los usuarios evaluar de forma aislada los efectos de una declaración o instrucción dada, el seguimiento de los efectos del cambio de la entrada, y obtener información sobre el comportamiento del programa (o mala conducta del mismo). Por otra parte, la fragmentación de la traza de ejecución puede identificar nuevas oportunidades de optimización del programa. Con esta metodología, un analista puede navegar, fragmentar, filtrar o buscar en la traza durante la ejecución del programa. El marco de análisis de trazas genérico se ha implementado en el sistema Anima y describimos una profunda evaluación experimental del marco que demuestra la utilidad del enfoque propuesto.

Resum

Els sistemes programari de hui en dia són artefactes complexos el comportament dels quals és sovint extremadament difícil d'entendre. Este fet ha portat al desenrotllament de sofisticades metodologies formals per a l'anàlisi, comprensió i depuració de programes.

L'anàlisi de traces d'execució consistix en tècniques que permeten buscar dinàmicament contingut específic en les traces d'execució d'un cert programa. La busca pot dur-se a terme cap avant o cap arrere. Si bé l'anàlisi cap avant es traduïx en una forma d'anàlisi sobre l'impacte que identifica l'abast i les possibles conseqüències dels canvis en l'entrada del programa, l'anàlisi cap arrere permet dur a terme l'anàlisi de la procedència; és a dir, mostra com (parts de) una eixida del programa depén de (parts de) la seua entrada i ajuda a estimar quines dades d'entrada són necessaris modificar per a dur a terme un canvi en el resultat.

En esta tesi, s'investiga una sèrie de metodologies d'anàlisi de traces que són especialment adequats per a l'anàlisi de traces d'execució llargues i complexes en la lògica de reescriptura (RWL) , que és un marc lògic i semàntic especialment adequat per a la formalització de sistemes altament concurrents i complexos.

La primera part de la tesi se centra a desenrotllar una tècnica (basada en la fragmentació incremental) d'anàlisi de traces cap arrere que aconseguix enormes reduccions en la grandària de la traça. Esta metodologia afavorix una millor anàlisi i depuració ja que la majoria de les inspeccions tedioses i irrellevants que es realitzen rutinàriament en el diagnòstic i la localització d'errors es poden eliminar de forma automàtica. Esta tècnica s'il·lustra per mitjà de diversos exemples que executem per mitjà del sistema iJulienne, una ferramenta interactiva de fragmentació que hem desenrotllat i que implementa la tècnica d'anàlisi de traces cap arrere.

En la segona part de la tesi es formalitza un sistema paramètric, ric i molt dinàmic, per a l'exploració de computacions en lògica de reescriptura. L'esquema implementa un algoritme d'animació genèric que permet l'execució indeterminista d'una donada teoria de reescriptura

condicional que serà objecte de seguiment per mitjà de l'ús de diferents modalitats, incloent una execució gradual pas a pas i una fragmentació automàtic cap avant i cap arrere, la qual cosa redueïx dràsticament la grandària i la complexitat de les traces baix inspecció i permeten als usuaris avaluar els efectes d'una donada declaració o instrucció de forma aïllada, el seguiment dels efectes del canvi de l'entrada, i obtindre informació sobre el comportament del programa (o mala conducta). D'altra banda, la fragmentació de la traça d'execució pot exposar les oportunitats d'optimització del programa. Amb esta metodologia, un analista pot navegar, fragmentar, filtrar o buscar en la traça durant l'execució del programa. El marc d'anàlisi de traces genèrica s'ha implementat en el sistema Anima i una profunda avaluació experimental reporta la utilitat de l'enfocament proposat.

Contents

Introduction **1**
Contributions of the Thesis 3
Part I – Backward Trace Analysis 3
Part II – Forward Trace Analysis 5
Related Work 7

0 Preliminaries **11**
0.1 The Term-language of Maude 11
0.2 Program Equations and Rules 13
0.3 Conditional Rewrite Theories 14
0.4 Rewriting in Conditional Rewrite Theories 18
0.5 Instrumented Execution Traces 21
0.6 Term Slices and their Concretizations 25
0.7 (Instrumented) Trace Slices and their Concretizations . . 27
0.8 Meaningful Descendants and Ascendants 28

I Backward Trace Analysis **31**

1 Backward Trace Slicing for Conditional Rewrite Theories **33**
1.1 Backward Slicing for Execution Traces 34
1.2 The Function *slice-step* 39
1.3 Correctness of Backward Trace Slicing 45

2 The *i*JULIENNE System **51**
2.1 *i*JULIENNE at Work 53
2.1.1 Debugging Maude Programs with *i*JULIENNE . . 53
2.1.2 Trace Querying with *i*JULIENNE 61
2.1.3 Dynamic Program Slicing 68
2.2 Experimental Evaluation 70

II Forward Trace Analysis **73**

3 Exploring Conditional Rewriting Logic Computations **75**
3.1 The Generic Exploration Scheme 76
3.1.1 Inspecting the Instrumented Traces 77
3.1.2 Exploring the Instrumented Computation Tree Slices 79

4 Exploration Modalities **83**
4.1 Interactive Stepper 83
4.2 Partial Stepper 84
4.3 Stepper and Partial Stepper Correctness 87
4.4 Forward Trace Slicer 90
4.5 Forward Trace Slicer Correctness 98
4.6 Backward Trace Slicing as an Instance of the Generic Scheme 101

5 The Anima system **105**
5.1 The Anima Exploration Tool 105
5.2 Implementation of the Tool 111

Conclusions **117**

Bibliography **121**

A Maude Specification of the Experimental Evaluation Examples **133**

List of Figures

1	Maude specification of the maze game	16
2	The 5×5 grid encoded in `MAZE`	17
3	Meaningful descendants of a rewrite step $s \stackrel{r,\sigma,w}{\to}_K t$	28
1.1	The `_mod_` operator	34
1.2	Maude specification of a distributed banking system	38
1.3	Backward step slicing function	39
1.4	Condition processing function	41
2.1	*i*JULIENNE architecture	52
2.2	Faulty Maude specification of a distributed banking system	54
2.3	*i*JULIENNE output for the trace $\mathcal{T}_{bank}$ w.r.t. the slicing criterion `ac( • ,-11)`	55
2.4	`BLOCKS-WORLD` faulty Maude specification	57
2.5	Program slice computed w.r.t. the slicing criterion `empty & empty`	58
2.6	Navigation through the trace slice of the *Blocks World* example	60
2.7	Navigation though the refined trace slice of the *Blocks World* example	60
2.8	Program slice computed w.r.t. the slicing criterion `hold( • )`	61
2.9	Maude specification of the `minmax` function	63
2.10	*i*JULIENNE output for the trace $\mathcal{T}^{\bullet}_{minmax}$ w.r.t. the slicing criterion automatically inferred in the query `PAIR(?,_)`	64
2.11	Loading the webmail execution trace	65
2.12	Querying the webmail trace w.r.t. the query `B(idA,_,?,_,_,_,_,_,_)`	66
2.13	Webmail trace slice after querying the trace	67
2.14	`MINMAX` program slice computed w.r.t. the query `PAIR(?,_)`	69

3.1 Computation tree 76
3.2 The inference rule of the transition system $(Conf, \Longrightarrow)$. 78
3.3 The one-step *expand* function 80
3.4 The interactive *explore* function 81

4.1 Inspection of the state s_0 w.r.t. $\mathcal{I}_{step}$ 84
4.2 Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{pstep}$ . . . 86
4.3 Inspection function that models the forward slicing of a conditional rewrite step 91
4.4 The condition processing function 92
4.5 Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{slice}$ 97

5.1 Anima architecture 106
5.2 Anima at work 107
5.3 Anima search mechanism 108
5.4 Anima trace information 109
5.5 Inspection of a condition with Anima 111
5.6 Benchmark problem for the `metaReducePath` command . 116

Introduction

The analysis of execution traces plays an important role in many program analysis techniques. Software systems commonly generate large and complex execution traces, whose analysis (or even their simple inspection) is extremely time-consuming and, in some cases, unfeasible to perform without adequate tool support. Existing tools for analyzing large execution traces usually rely on a set of visualization techniques that facilitate the exploration of the trace content. Common capabilities of these tools include the option to simplify the traces by hiding some specific contents and to step the program execution while searching for particular components. Nearly all modern IDEs, debuggers, and testing tools currently support this mode of execution optionally, where animation or *stepping* is typically achieved either by forcing execution breakpoints, instruction simulation, or code instrumentation.

Rewriting Logic is a very general *logical* and *semantic framework* that is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [BBF09] and Web systems [ABFR06, ABR09, ABER10, ABF+13, ABR14]). Rewriting Logic is efficiently implemented in the high-performance system Maude [CDE+11]. Roughly speaking, a *rewriting logic theory* seamlessly combines a *term rewriting system* (TRS) with an *equational theory* that may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are performed *modulo* the equations and axioms. Rewriting logic-based tools, like the Maude-NPA protocol analyzer [EMM06], the Maude LTLR model checker [BM12], the Java PathExplorer runtime verification tool [HR01], and the WifiX tool for repairing XML repositories [ABF+13] (just to mention a few [MOPV12]), are used in the analysis and verification of programs, systems, documents and applications wherein the system states are represented as algebraic entities (elements of an algebraic data type that is specified by the equational theory), while the system computations are modeled via rewrite rules, which describe transitions between states and are performed *modulo* the equations and axioms. The execution traces produced by these tools are

usually very complex and are therefore not amenable to manual inspection. However, not all the information that is in the trace is needed to analyze a given piece of information in a target state and the trace can be simplified by focusing on some particular contents, which favors better analysis and monitoring [CR09]. Trace slicing is an automated transformation technique that simplifies the computation trace by removing some irrelevant components that we do not want to observe. Trace slicing is helpful for program debugging, improvement, and understanding [ABE+11, FR01]. For instance, consider the following rules that define in Maude (a part of) the natural semantics of a simple imperative language:

1)
```
crl <while B do I, St> => <skip, St>
if <B, St> => false /\ isCommand(I)
```

2)
```
rl <skip, St> => St
```

3)
```
rl <false, St> => false
```

Then, in the two-step execution trace `<while false do X := X + 1, {}>` $\rightarrow$ `<skip, {}>` $\rightarrow$ `{}`, we can observe that the statement `X := X + 1` is not relevant to compute the output `{}`. Therefore, the trace could be simplified by hiding or removing `X := X + 1`.

Debugging and optimization techniques based on RWL have received growing attention [ABBF10, ABE+11, MOM02, RVCMO09, RVMO10b], but to the best of our knowledge, no trace analysis tool for conditional RWL theories has been formally developed to date. To debug Maude programs, Maude has a basic tracing facility that allows the user to advance through the program execution stepwisely with the possibility to set break points, and lets him/her select the statements to be traced, except for the application of algebraic axioms that are not under user control and are never recorded explicitly in the trace. All rewrite steps that are obtained by applying the equations or rules for the selected function symbols are shown in the output trace so that the only way to simplify the displayed view of the trace is by manually fixing the traceable equations or rules. Thus, the trace is typically huge and incomplete, and when the user detects an erroneous intermediate result, it is difficult to determine where the incorrect inference started. Moreover, this trace is either directly displayed or written to a file (in both cases, in plain text format) thus only being amenable for manual inspection by the

user. This is in contrast with the enriched traces described in this work, which are complete (all execution steps are recorded by default) and can be automatically simplified in order to facilitate a specific analysis. Also, the trace can be directly displayed or delivered in its meta-level representation, which is very useful for further automated manipulation.

Contributions of the Thesis

This thesis is organized in two parts. Part I presents the first slicing-based, backward trace analysis methodology for conditional RWL computations [ABFR12a, ABFR12b], together with an incremental version of the technique and a practical tool that implements the method [ABFS13b]. Our proposal for trace slicing is aimed at endowing the RWL framework with a new instrument that can be used to analyze Maude execution traces and improve Maude programs. Part II defines a semantics-driven, parameterized trace exploration technique for conditional RWL computations [ABFS13a, ABFS15, ABFS14] that supports both, backward and forward analysis of the computation trace.

The proposed methodologies are fully general and can be applied for debugging as well as for optimizing any RWL-based tool that produces or manipulates RWL computations (i.e., Maude execution traces).

In the following, we briefly summarize the main contributions of the two parts of this thesis.

Part I – Backward Trace Analysis

This part offers an up-to-date, comprehensive, and uniform presentation with examples of the backward trace analysis methodology based on the trace slicing technique developed in [ABE+11, ABER11, ABFR12a, ABFR12b] and a totally redesigned implementation of the conditional trace slicer *i*JULIENNE [ABFS13b].

The contributions of Part I can be summarized as follows:

1. We describe a novel slicing technique for Maude programs that can be used to drastically reduce complex, textually-large system computations w.r.t. a user-defined slicing criterion that selects those

data that we want to track back from a given point. The distinguishing features of our technique are as follows:

(a) The technique copes with the most prominent RWL features, including algebraic axioms such as associativity, commutativity, and unity.

(b) The technique also copes with the rich variety of conditions that can occur in Maude theories (i.e., equational conditions $s = t$, matching conditions $p := t$, and rewrite expressions $t \Rightarrow p$) by taking into account the precise way in which Maude mechanizes the conditional rewriting process so that all such rewrite steps are revisited backwards in an instrumented, fine-grained way.

(c) Unlike previous backward tracing approaches, which are based on a costly, dynamic labeling procedure [ABER11, TeR03], here a less expensive, incremental technique of matching refinement based on unification is used that allows the relevant data to be traced back.

2. We have developed the iJULIENNE system, which is the first slicing-based, incremental trace analysis tool that can be used to analyze execution traces of RWL-based programs and tools. iJULIENNE supersedes and greatly improves the preliminary system presented in [ABFR12b]. The original algorithm was developed under the assumption that the user examines and slices the entire trace w.r.t. a static slicing criterion, in a non-incremental way. In contrast, the slicing criterion in iJULIENNE can be dynamically revised, in order to achieve further simplifications. The main features provided by the iJULIENNE trace analyzer are listed below.

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

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

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

3. To give the reader a better feeling of the generality and wide application range of our conditional slicing approach, we describe some applications of *i*JULIENNE to debugging, analysis and improvement of complex systems (e.g., biological systems, web systems, and protocol specifications).

Part II – Forward Trace Analysis

This part presents the first forward trace analysis methodology for RWL theories. This methodology is formalized as an instance of a parameterized scheme for exploring RWL computations in conditional rewrite theories that involve rewriting modulo associativity (A), commutativity

(C), and unity (U) axioms. A program animator and a partial stepper are also formulated as instances of the generic scheme.

The contributions of Part II can be summarized as follows:

1. We formulate a generic animation algorithm that can be tuned to work with different modalities, including *incremental stepping*, *partial stepping* and *automated forward/backward slicing*.

2. The forward slicing technique allow us to inspect the execution trace with regard to a set of symbols of interest (input symbols), so that, all data that are not descendants of the observed symbols are filtered out.

3. The partial stepper supports a form of abstract computation that allows computations with partial inputs to be stepped.

4. The generic scheme is implemented and tested in a graphical tool called Anima, which provides a mighty and versatile environment for the dynamic analysis of RWL computations. The main features of Anima are listed below:

    (a) Anima is equipped with three inspection modalities including interactive stepping, partial stepping and forward slicing.

    (b) The user can freely display either a simplified view of a rewrite step or the complete and detailed sequence of steps in a corresponding instrumented trace that we construct. The instrumentation allows the relevant information of the rewrite steps to be traced explicitly despite the fact that terms are rewritten modulo a set of axioms that may cause their components to be implicitly reordered.

    (c) A search facility is implemented where a pattern language allows the selected information of interest to be searched in huge states of complex computation trees. The user provides a filtering pattern (the query) that specifies the set of symbols that he/she wants to search for, and then all the states matching the query are automatically highlighted in the computation tree.

(d) Anima facilitates the inspection of the conditions satisfied during the application of a conditional rule or equation, which allows the user to export the traces deployed by evaluating the conditions to *i*Julienne for further analysis.

(e) The system integrates the backward trace analyzer *i*Julienne [ABFR14] into Anima, thus enabling the possibility to explore computations both back and forth in order to validate input data or to locate programming mistakes. Anima is endowed with the very same powerful and intuitive Web user interface of *i*Julienne.

Related Work

Dependency analysis techniques provide a formal foundation for forms of provenance and impact analysis that are intended to highlight parts of the program input (resp. output) on which a part of its output (resp. input) depends [CAA11]. This is essentially achieved by computing the symbol dependencies across computations and has proved to be useful in many contexts such as efficient memorization and caching [ALL96], aiding program debugging via unconditional slicing [ABER11, FT94], information-flow security [SM03] and several other forms of program analysis techniques, just to mention a few. A great deal of work has been done on each of these topics, and we refer to [CAA11] for further references that we cannot survey here.

The notion of descendants [Klo90] or residuals[1] [HL79, O'D77] with its inverse notion of ancestors or origins is classical in the theory of rewriting, both in first-order term rewriting and in higher-order rewriting, such as lambda calculus. While dependency provenance provides information about the origins of (or influences upon) a given result, the notion of descendants is the key for impact evaluation, that is, to assess the changes that can be attributed to a particular input or intervention [BKV00]. For orthogonal term rewriting systems (i.e., left–linear and overlap–free), a refined version of the descendant/ancestor relation, based on the notion of symbol tracking, was first introduced in [Klo90]. Several variants

[1] In the literature, the term 'residual' is usually reserved for a descendant of a redex [BKV00].

of this notion have been studied, sometimes with applications that are similar to the ones described in this thesis (see [BKV00] for references). Nonetheless, non-left linear and collapsing rules are not considered or are dealt with using ad-hoc strategies in these works, while our approach requires no special treatment of such rules. An extension of [Klo90] for all TRSs is described in [TeR03] and a method for implementing origin tracking in conditional term rewriting systems is given in [DKT93].

This thesis describes forward and backward trace slicing techniques that are suitable for provenance and impact analysis in conditional rewrite theories. Furthermore, the proposed techniques simplify and extend the formal development in [ABER11] by getting rid of the complex dynamic labeling algorithm that was needed to trace back the origins of the symbols of interest, replacing it with a simple unification mechanism that allows control and data dependencies to be propagated back and forth between consecutive rewrite steps. Our techniques also avoid manipulating the ascendants and descendants by recording their addressing positions; we simply and explicitly record the meaningful positions within the computed term slices themselves, without resorting to any other artifact. Moreover, our slicing techniques can take advantage of the filtered information for the purpose of dynamic program slicing, that is, the computation of the set of program statements, that may affect the values at some point of interest.

A generic, static technique to infer (forward) program slices is defined in [RAA13]. This technique relies on the formal executable semantics of the language of interest, which is given as a RWL theory. Informally, this kind of slices represent program fragments that are affected by a particular program statement with respect to a set of variables of interest. Different from our methodologies, this technique is static and defines program slices (rather than trace ones) by using (meta-level) over-approximation.

On the practical side, we are not aware of any *trace slicer* that is comparable to *i*JULIENNE for either imperative or declarative languages. To the best of our knowledge, there are just a couple of slicers that only slightly relate to ours.

HaSlicer [RB05] is a prototype of a slicer for functional programs written in Haskell that is used for the identification of possible coherent components from (functional) monolithic legacy code. Both backward and forward dependency slicing are covered by HaSlicer, which is proposed as

a support tool for the software architect to manually improve program understanding, and automatically discover software components. The latter is particularly useful as an architecture understanding technique in earlier phases of a re-engineering process.

Spyder [ADS93] is a prototype debugger for C that, thanks to the combination of dynamic *program slicing* and execution backtracking techniques, is able to automatically step-back, statement by statement, from any desired location in order to determine which statements in the program affect the value of an output variable for a given test case, and to determine the value of a given variable when the control last reached a given program location.

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

The Haskell interactive debugger Hat [CRW00] also allows execution traces to be explored backwards, starting from the program output or an error message (computation abort). Similarly to Spyder, this task is carried out by navigating a graph-like, supplementary data structure (redex trail) that records dependencies among function calls. Even if Hat is able to highlight the top-level "parent" function of any expression in the final trace state, it cannot be used to compose a full trace slice. Actually, at every point of the recreated trail, the function arguments are shown in fully evaluated form (the way they would appear at the end of the computation) even though, at the point at which they are shown, they would not yet have necessarily reached that form. A totally different, bytecode trace *compression* was proposed in [jsl08] to help perform Java program analysis (e.g., dynamic *program slicing* [Tip95]) over the compact representation.

Finally, an algebraic stepper for Scheme is defined and formally proved in [CFF01], which is included in the DrScheme programming environment. In order to discover all of the steps that occur during the program evaluation, the stepper rewrites (or "instruments") the code. This is in contrast to our stepper Anima, which does not rely on program instrumentation.

Chapter 0

Preliminaries

In this chapter, we recall some basic notions that will be used in the rest of the thesis. We assume some basic knowledge of term rewriting [TeR03] and Rewriting Logic [Mes92]. Some familiarity with the Maude language [CDE+11, CDE+07] is also required.

Maude is a rewriting logic [Mes92] specification and verification system whose operational engine is mainly based on a very efficient implementation of rewriting. Maude's basic programming statements are equations and rules. Equations are used to express deterministic computations, which lead to a unique final result, while rules naturally express concurrent, nondeterministic, and possibly nonterminating computations. A Maude program that contains only equations (together with the syntax declaration for sorts, operators, and variables) is called a functional module and essentially defines one or more functions by means of equations. A Maude program that contains rules and possibly equations is called a system module, where the rules define transitions in a likely concurrent system. Maude notation will be introduced "on the fly" as required.

0.1 The Term-language of Maude

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [CDE+11]. The connected components of $(S, <)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_<$ containing $<$. We assume an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s\in S}$ of disjoint variable sets. $\tau(\Sigma, \mathcal{V})_s$ and $\tau(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. We write $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ for the corresponding term algebras. A simple syntactic condition on Σ and $(S, <)$, called *preregularity* [CDE+11], ensures that each (well-formed) term t always has a least-sort possible among all sorts

in S, which is denoted $ls(t)$. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. In order to simplify the presentation, we often disregard sorts when no confusion can arise.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Positions are ordered by the prefix ordering, that is, given the positions w_1 and w_2, $w_1 \leq w_2$ if there exists a position u such that $w_1.u = w_2$. Given two positions w_1 and w_2, we say that w_1 and w_2 are not comparable iff $w_1 \not\leq w_2$ and $w_2 \not\leq w_1$. Given a set of positions P, and a position $p \in P$, we say that p is *minimal* w.r.t. P, iff there does not exist a position $p' \in P$ such that $p' \leq p$ and $p' \neq p$ (i.e., p' is strictly smaller than p). By $t|_w$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of *replacing the subterm* $t|_w$ by the term s in t.

A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $\tau(\Sigma, \mathcal{V})$, which is equal to the identity almost everywhere except over a set of variables $\{x_1, \ldots, x_n\}$. The *domain* of σ is the set $Dom(\sigma) = \{x \in \mathcal{V} \mid x\sigma \neq x\}$. By ε, we denote the *identity* substitution. The application of a substitution σ to a term t, denoted $t\sigma$, is defined by induction on the structure of terms:

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

Given two terms s and t, a substitution σ is the *matcher* of t in s, if $s\sigma = t$. The term t is an *instance* of the term s (in symbols, $s \leq t$), iff there exists a matcher σ of t in s. By $match_s(t)$, we denote the function that returns a matcher of t in s if such a matcher exists. Given two substitutions θ and θ', their *composition* $\theta\theta'$ is defined as $t(\theta\theta') = (t\theta)\theta'$ for every term t. We recall that composition is associative. A substitution σ is more general than θ, denoted by $\sigma \leq \theta$, if $\theta = \sigma\gamma$ for some substitution γ. We say that a substitution σ is a *unifier* of two terms t and t' if $t\sigma = t'\sigma$. We let $mgu(t, t')$ denote a *most general unifier* σ of t and t' (i.e., $\sigma \leq \theta$ for any other unifier θ of t and t'). Let the *parallel composition* $\psi_1 \Uparrow \psi_2$ of substitutions be defined as in [Pal90, Hof11]. We

compute the parallel composition $\psi_1 \Uparrow \psi_2$ of two substitutions ψ_1 and ψ_2 as follows:

$$\begin{array}{rl}\psi_1 \Uparrow \psi_2 = & mgu(f(x_1,\ldots,x_n,y_1,\ldots,y_m),\\ & f(x_1\psi_1,\ldots,x_n\psi_1,y_1\psi_2,\ldots,y_m\psi_2))\end{array}$$

where $x_i, i \in \{1,\ldots,n\}$, and $y_j, j \in \{1,\ldots,m\}$, are the domain variables of ψ_1 and ψ_2, respectively, and f is a function symbol of $n+m$-arity.

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 a binary relation $\rightarrowtail$, we define the usual *transitive* (resp., *transitive and reflexive*) closure of $\rightarrowtail$ by $\rightarrowtail^+$ (resp., $\rightarrowtail^*$).

0.2 Program Equations and Rules

Our techniques in this thesis deal with conditional RWL theories. We consider three different kinds of conditions that may appear in a conditional Maude theory: an *equational condition*[1] e is any (ordinary) equation $t = t'$, with $t, t' \in \tau(\Sigma,\mathcal{V})$; a *matching condition* is a pair $p := t$ with $p, t \in \tau(\Sigma,\mathcal{V})$; a *rewrite expression* is a pair $t \Rightarrow p$, with $p, t \in \tau(\Sigma,\mathcal{V})$.

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

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

[1]A Boolean equational condition $b = \mathit{true}$, with $b \in \tau(\Sigma,\mathcal{V})$ of sort Bool is simply abbreviated as b. A *Boolean condition* is a conjunction of abbreviated Boolean equational conditions.

sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is an equational condition, a matching condition, or a rewrite expression. Unlike matching conditions, which can only use equations to evaluate the input term t, rewrite expressions can apply both equations and rewrite rules for the evaluation. When the condition C is empty, we simply write $[l] : \lambda \Rightarrow \rho$ and use the keyword `rl` to declare it in Maude. A conditional rule $[l] : \lambda \Rightarrow \rho \textit{ if } c_1 \wedge \ldots \wedge c_n$ is *admissible* iff it fulfills the exact analogy of the admissibility constraints (i) and (ii) for the equational conditions and the matching conditions, plus the following additional constraint: for each rewrite expression c_i in C of the form $e \Rightarrow p$, $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$.

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

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

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

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

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

0.3 Conditional Rewrite Theories

Roughly speaking, a (conditional) rewrite theory [Mes92] seamlessly combines a set of conditional rewrite rules (or conditional term rewriting system, CTRS), with an equational theory (also possibly conditional) that

may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are applied modulo the equations and axioms. Within this framework, the system states are typically represented as elements of an algebraic data type that is specified by the equational theory, while the system computations are modeled via the rewrite rules, which describe transitions between states.

More formally, an *order-sorted equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of (oriented) admissible, conditional equations, and B is a collection of unconditional equational axioms (e.g., associativity, commutativity, and unity) that can be associated with any binary operator of Σ. The equational theory E induces a congruence relation on the term algebra $\tau(\Sigma, \mathcal{V})$, which is denoted by $=_E$.

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

Example 0.3.1

Consider the Maude system module `MAZE` of Figure 1, which is inspired by the maze example in [RVMOC12]. The module is delimited by the Maude keywords `mod` and `endm`, and it encodes a conditional rewrite theory that specifies a maze game in which multiple players (modeled as terms of sort `Player`) must reach a given exit point. Players may enter the maze at distinct entry points, and can move through the maze by walking or jumping. Furthermore, any collision between two players eliminates them from the game.

`MAZE` imports and makes use of the predefined Maude module `NAT`, which provides the equational definition for natural numbers together with some common built-in operators for their manipulation such as addition (`+`) and subtraction[3] (`sd`). The operators in the module signature

[2]Equational specifications in Maude can be theories in membership equational logic, which may include conditional membership axioms that are not addressed in this thesis. Actually, membership axioms can interact with operator attributes such as `assoc` and `iter` in undesirable ways [CDE+11], which can be a major difficulty for a tracing-based tool like ours to work correctly.

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

```
mod MAZE is
  pr NAT .

  sorts Pos List State Player .
  subsort Pos < List .

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

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

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

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

Figure 1: Maude specification of the maze game.

are declared using the keyword `op`, while their types structure is specified using the keywords `sorts` and `subsorts`. Module variables are declared by means of the keyword `vars`. Roughly speaking, a maze is a `size` $\times$ `size` grid in which each maze position is specified by a pair of natural numbers `< X,Y >` of sort `Pos`.

The internal maze structure is defined through the equation `wall`,

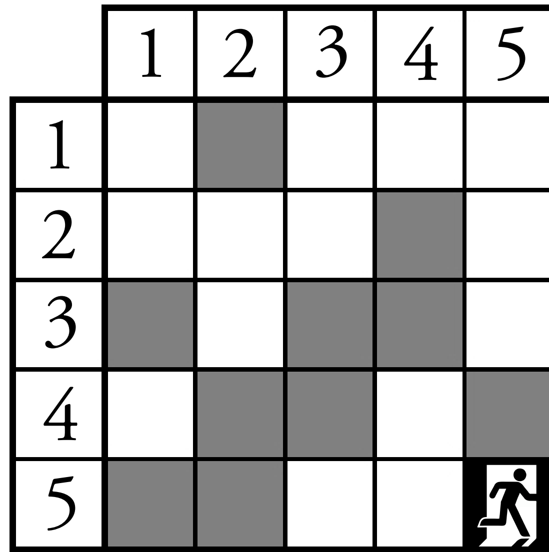


Figure 2: The 5×5 grid encoded in `MAZE`.

which explicitly defines those cells that represent the maze walls (see Figure 2). Each player's path[4] in the maze is described by a term of sort `List` that specifies a list of (pairwise distinct) positions by means of the usual constructor operator `nil` (empty list) and the associative, juxtaposition operator `__` whose unity element is `nil`.

System states describe a game scenario by recording the paths taken by the different players from the moment they entered the game (at their respective entry points). We use the associative and commutative operator `||` (whose unity element is the constant `empty`) to model states as multisets of triples of the form $\{p_1, L_1, m_1\}$ `||` $\ldots$ `||` $\{p_n, L_n, m_n\}$, $p_i \neq p_j$ for $i \neq j$, where p_i uniquely identifies the `i`th player, L_i is the path the `i`th player has hitherto followed since he entered the maze, and m_i is the length of the path L_i (which is different from the length of the list because players are allowed to jump two boxes in a single move), with

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

`i` $=$ `1`$, \ldots,$ `n`.

Given a player's path `L`, the next possible player's moves are non-deterministically computed by the rules `walk` and `jump`, which respectively augment `L` with the position `P` delivered by the rewrite expressions `next(L,N) => P`, with `N` $=$ `1` (`walk`) or `N` $=$ `2` (`jump`), occurring in the condition of these two rules. The function `next(L,N)` models all the possible `N`-cell movements that are available from the current player's location (given by the last position in `L`). In both rules, the correctness of the computed subsequent position `P` is verified by means of the function `isOK(L P)`. Specifically, position `P` is valid iff it is within the limits of the maze, not repeated in `L`, and not a part of the maze wall. Note that the `jump` rule allows a player to leap over either a wall or another player provided the position reached is valid.

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

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

Finally, note that the `exit` and `eject` operations should be modeled by using equations rather than rewrite rules in order to provide appropriate, deterministic `exit` and `eject` behavior. Nonetheless, we deliberately specified them by using rules in order to illustrate the debugging capabilities of our forward exploration technique in Chapter 4.

The slicing techniques formalized in this thesis are formulated by considering the precise way in which Maude proves the conditional rewriting steps modulo an equational theory $E = \Delta \cup B$, which we describe in the following section (see Section 5.2 in [CDE+11] for more details).

0.4 Rewriting in Conditional Rewrite Theories

Given a conditional rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the conditional rewriting modulo E relation (in symbols, $\to_{R/E}$) 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]. In other words, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is, in general, undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

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

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

Since equations and axioms are both interpreted as rewrite rules in our formulation for specific purposes, notation $\lambda \Rightarrow \rho \text{ } \mathit{if} \text{ } C$ is often abused throughout this thesis to denote rewrite rules as well as (oriented) equations and axioms.

Roughly speaking, a conditional rewrite step on the term t applies a rewrite rule/equation to t by replacing a *red*ucible (sub-)*ex*pression of t (namely $t_{|w}$), called the *redex*, by its contracted version $\rho\sigma$, called the *contractum*, whenever the condition C is fulfilled. Note that the evaluation of a condition C is typically a recursive process since it may involve further (conditional) rewrites in order to normalize C to *true*. Specifically, an equational condition e *evaluates to true* w.r.t. σ if $e\sigma\downarrow_{\Delta,B} =_B \mathit{true}$;

a matching equation $p := t$ *evaluates to true* w.r.t. σ if $p\sigma =_B t\sigma{\downarrow_{\Delta,B}}$; a rewrite expression $t \Rightarrow p$ *evaluates to true* w.r.t. σ if there exists a rewrite sequence $t\sigma \rightarrow^*_{R\cup\Delta,B} u$, such that $u =_B p\sigma$.[5] Although rewrite expressions and matching/equational conditions can be intermixed in any order, we assume that their satisfaction is attempted sequentially from left to right, as in Maude.

Under appropriate conditions on the rewrite theory, a rewrite step $s \rightarrow_{R/E} t$ modulo E on a term s can be implemented without loss of completeness by applying the following rewrite strategy [DM10]:

1. **Equational simplification of s in Δ modulo** B, that is, reduce s using $\rightarrow_{\Delta,B}$ until the canonical form w.r.t. Δ modulo B ($s\downarrow_{\Delta,B}$) is reached;

2. **Rewrite** ($s\downarrow_{\Delta,B}$) **in** R **modulo** B to t' using $\rightarrow_{R,B}$, where $t' \in [t]_E$.

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

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

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

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

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

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

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

Example 0.4.1

Consider the rewrite theory of Example 0.3.1. A Maude step for the initial term `next(< 1,1+1 >,1)` is as follows:

$$
\begin{aligned}
m\mathcal{S}(\texttt{next(< 1,1+1 >,1)}) = \;& \texttt{next(< 1,1+1 >,1)} \xrightarrow{\texttt{builtIn}(+)}_{\Delta,B} \\
& \texttt{next(< 1,2 >,1)} \xrightarrow{\texttt{downN}}_{R,B} \\
& \texttt{< 1,2+1 >} \xrightarrow{\texttt{builtIn}(+)}_{\Delta,B} \\
& \texttt{< 1,3 >}
\end{aligned}
$$

Note that since built-in operators are not provided in Maude with an explicit rule-based specification, we handle them in a special way. Given a built-in operator `op` and an execution trace $\mathcal{C}$, the idea is to handle every reduction `a op b` $\to$ `c` that occurs in $\mathcal{C}$ as an ordinary rewrite step, which is done by adding the extra equation `a op b` $=$ `c` to the considered equational theory. This way, every application of `op` that occurs in $\mathcal{C}$ is mimicked by applying its corresponding built-in equation.

0.5 Instrumented Execution Traces

In this section, we introduce an auxiliary technique for instrumenting execution traces. The instrumentation allows the relevant information of the rewrite steps, such as the selected redex and the contractum produced by the step, to be traced explicitly despite the fact that terms are rewritten modulo a set B of equational axioms that may cause their components to be implicitly reordered. Given an execution trace $\mathcal{C}$, let us show how $\mathcal{C}$ can be expanded into an *instrumented* execution trace $\mathcal{T}$ in which each application of the matching modulo B algorithm that is used in $\to_{R,B}$-steps and $\to_{\Delta,B}$-steps is explicitly mimicked by the specific

application of a bogus equational axiom, which is oriented from left to right and then applied as a rewrite rule in the standard way.

Typically hidden inside the B-matching algorithms, some pertinent term transformations allow terms that contain operators obeying equational axioms to be rewritten into supportive B-normal forms that facilitate the matching modulo B. In the case of AC-theories, these transformations allow terms to be reordered and correctly parenthesized to enable subsequent rewrite steps. Basically, this is achieved by producing a single, auxiliary representative of their AC congruence class (i.e., the AC-normal form). An AC-normal form is typically generated by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, sorting these arguments under some linear ordering and combining equal arguments using multiplicity superscripts [Eke03]. For example, the congruence class containing $f(f(\alpha, f(\beta, \alpha)), f(f(\gamma, \beta), \beta))$ where f is an AC symbol and subterms α, β, and γ belong to alien theories might be represented by $f^*(\alpha^2, \beta^3, \gamma)$, where f^* is a variadic symbol that replaces nested occurrences of f. A more formal account of this transformation is given in [Eke95].

As for purely associative theories, we can get an A-normal form by just flattening nested function symbol occurrences without sorting the arguments. This case has practical importance because it corresponds to lists. C-normal forms are just obtained by properly ordering the arguments of a commutative binary operator. Finally, for function symbols that satisfy the unit axiom U, the unity element of U is not included in the U-normal form, and variables under a U symbol can always be assigned the unity element through U-matching [Eke95].

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

In the following, we discuss how we can simulate B-matching in our framework by means of specific "fake" axioms that mimick the B-matching transformation of terms that occur internally in Maude. This allows these transformations to be unhidden and explicitly revealed in the output instrumented trace. This artifice is only a means to reveal

the term transformations of subterms that are forced by the step so that any position can be properly traced across rewriting steps.

Example 0.5.1

Consider a binary AC operator f together with a simple, standard lexicographic ordering over constant symbols. Given the term $f(b, f(f(b, a), c))$, let us reveal how this term matches modulo AC the left-hand side of the rule $[r] : f(f(x, y), f(z, x)) \Rightarrow x$ with AC-matching substitutions $\{x/b, y/a, z/c\}$ and $\{x/b, y/c, z/a\}$. For the first solution, this is mimicked by the transformation sequence $f(b, f(f(b, a), c)) \stackrel{\texttt{toACnf}}{\longrightarrow} f^*(a, b^2, c)$ $\stackrel{\texttt{fromACnf}}{\longrightarrow} f(f(b, a), f(c, b))$, where 1) the first step corresponds to a term transformation that obtains the AC-normal form $f^*(a, b^2, c)$, and 2) the second step corresponds to the inverse, an unflattening transformation that delivers the AC-equivalent term $f(f(b, a), f(c, b))$ that syntactically matches the left-hand side of rule r with substitution $\{x/b, y/a, z/c\}$. Note that an alternative unflattening transformation is possible $f^*(a, b^2, c) \stackrel{\texttt{fromACnf}}{\longrightarrow} f(f(b,\ c), f(a, b))$, which actually delivers the second AC-matcher $\{x/b, y/c, z/a\}$. When several B-matchers exist, we only consider those that are effectively computed by means of the Maude internal rewriting strategy.

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

Example 0.5.2

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

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

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

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

Example 0.5.3

Consider the rewrite theory of Example 0.3.1 together with the following execution trace that consists of a single Maude step, which makes a downward movement from the current position modeled by the one-element list `< 1,1 >`:

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

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

$$\begin{aligned}\mathcal{T} = \; & \texttt{next(< 1,1 >,1)} \overset{\texttt{fromAUnf}}{\longrightarrow}_{B} \texttt{next(nil < 1,1 >,1)} \overset{\texttt{downN}}{\longrightarrow}_{R} \\ & \texttt{< 1,1+1 >} \overset{\texttt{builtIn}(+)}{\longrightarrow}_{\Delta} \texttt{< 1,2 >}\end{aligned}$$

where the first, extra step

$$\texttt{next(< 1,1 >,1)} \overset{\texttt{fromAUnf}}{\longrightarrow}_{B} \texttt{next(nil < 1,1 >,1)}$$

has been added to describe the internal transformation that affixes the identity element `nil` of the AU list operator `__` to the one-element list `< 1,1 >`. This is simply formalized as a rewrite step by using the bogus axiom `rl [fromAUnf] : < 1,1 > => nil < 1,1 >`. This transformation enables the subsequent application of the rewrite rule `downN`.

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

In order to improve readability, we often omit B-matching transformations and the evaluation of built-in operators[6] when displaying Maude steps (unless explicitly stated otherwise). This is consistent with the strategy adopted by Maude for the case of B-matching transformations, and it is the default option in our slicing tool.

0.6 Term Slices and their Concretizations

A term *slice* of a term s is a term abstraction $s^\bullet$ that shields part of the information in s; that is, the irrelevant data in s that we are not interested in are simply replaced by special $\bullet$-variables of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$. More precisely, in our framework, the term slice $s^\bullet$ can be seen as the term s plus a mask that hides the irrelevant symbols of s via $\bullet$-variables. In this way, the irrelevant part of $s^\bullet$ can be conveniently recovered from $s^\bullet$ at any time by simply unhiding the masked symbols, which we formalize using the function $\mathit{unhide}(s^\bullet) = s$ for every term slice $s^\bullet$ of s. More formally, let $\mathcal{V}^\bullet$ be the extension of the S-sorted family $\mathcal{V}$ that augments $\mathcal{V}$ with $\bullet$-variables in the obvious way. A term $s^\bullet$ is a *term slice* of the term s iff s is an instance of $s^\bullet$ and $s^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$.

[6] Maude provides efficient (C-like) built-in operators such as the addition `(+)` of natural numbers. This is thanks to a built-in mechanism called `iter` (short for *iterated operator*), which permits the efficient manipulation of very large stacks of unary operators, and an efficient binary representation of unbounded natural number arithmetic [CDE+11].

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

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

Definition 0.6.1 (Term Slice) *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. Then, the term slice $\mathit{Tslice}(t, P)$ of t w.r.t. P is computed as follows.*

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

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

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

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

Example 0.6.2

Consider the specification of Example 0.3.1 and initial state:

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

Consider the set $P = \{1.1, 1.2.1, 1.2.2, 2.1\}$ of positions in t. Then, $\mathit{Tslice}(t, P) = \{\texttt{p1,< 4,4 >, }\bullet_1\ \} \texttt{ || } \{\texttt{p2}, \bullet_2, \bullet_3\}$ and the set of meaningful positions $\mathcal{MP}os(t^\bullet) = \{\Lambda, 1, 1.1, 1.2, 1.2.1, 1.2.2, 2, 2.1\}$.

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

Definition 0.6.3 (Term Slice Concretization) *Let $t, t' \in \tau(\Sigma, \mathcal{V})$ be two terms. Let $t^\bullet$ be a term slice of t. We say that t' is a* concretization *of $t^\bullet$, if there exists a substitution σ such that $t^\bullet\sigma = t'$.*

Example 0.6.4

Let $t^\bullet = \bullet_1 + \bullet_2 + \bullet_2$. Then, $10 + 2 + 2$ is a concretization of $t^\bullet$, while $10 + 2 + 3$ is not.

In the following section, the concretization of an (instrumented) trace slice is defined.

0.7 (Instrumented) Trace Slices and their Concretizations

An (instrumented) *trace slice* $\mathcal{T}^\bullet$ of an (instrumented) trace $\mathcal{T}$ is a kind of trace abstraction that shields part of the information in $\mathcal{T}$. Roughly speaking, given an instrumented execution trace $\mathcal{T} = s_0 \rightarrow_K s_1 \rightarrow_K \cdots \rightarrow_K s_n$, the instrumented trace slice has the form $\mathcal{T}^\bullet = s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet$ where each $s_i^\bullet \in \tau(\Sigma, \mathcal{V}^\bullet)$ is a term slice of s_i.

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

Definition 0.7.1 (Instrumented Trace Slice Concretization) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and $\mathcal{T} = (s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K s_1 \stackrel{r_2,\sigma_2,w_2}{\rightarrow}_K \cdots \stackrel{r_n,\sigma_n,w_n}{\rightarrow}_K s_n)$ be an instrumented execution trace in $\mathcal{R}$. Given the instrumented trace slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet)$ of $\mathcal{T}$, an instrumented trace slice* concretization *$\mathcal{T}'$ of $\mathcal{T}^\bullet$ w.r.t. $\mathcal{T}$ is any instrumented execution trace $s'_0 \stackrel{r_1,\sigma'_1,w_1}{\rightarrow}_K s'_1 \stackrel{r_2,\sigma'_2,w_2}{\rightarrow}_K \cdots \stackrel{r_n,\sigma'_n,w_n}{\rightarrow}_K s'_n$ such that each $s_i^\bullet$ is a term slice of $s'_i \in \tau(\Sigma, \mathcal{V})$, for all $i = 0, \ldots, n$.*

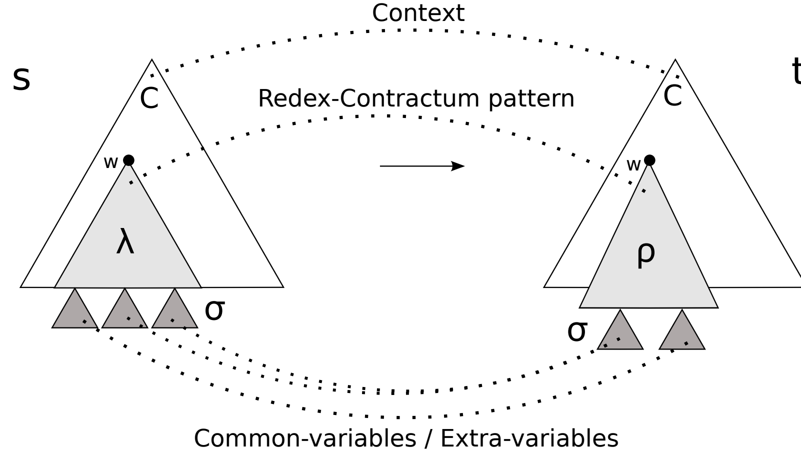


Figure 3: Meaningful descendants of a rewrite step $s \stackrel{r,\sigma,w}{\to}_K t$.

0.8 Meaningful Descendants and Ascendants

The trace slicing algorithms presented in this thesis enforce the notion of *meaningful* descendants and ascendants that we define as follows.

Given a conditional rewrite step $\mu = (s \stackrel{r,\sigma,w}{\to}_K t)$ and a term slice $s^\bullet$ of s, the computed term slice $t^\bullet$ that contains the meaningful descendants of $s^\bullet$ by $\mathcal{I}_{slice}$ w.r.t. the rewrite step μ and $s^\bullet$ is given by the following relations, which are illustrated in Figure 3.

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

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

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

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

The dual notion of *meaningful* ascendants can be defined in the obvious way by just inverting $s^{\bullet}$ and $t^{\bullet}$ in the meaningful descendant definition; that is, a symbol *sym* in $s^{\bullet}$ is a meaningful ascendant of a symbol *sym*′ in $t^{\bullet}$ iff *sym*′ is a meaningful descendant of *sym*. The following example illustrates the notion of meaningful descendant.

Example 0.8.1

Consider the conditional rewrite rule

```
crl [r] :  f(X,Y) => h(Z,X) if Z := g(X,Y)
```

that contains an extra-variable `Z` in its right-hand side, together with the equational definition `eq [sum] :  g(X,Y) = X + Y`. Let us consider the one-step trace $\mathcal{T}$ = `c(f(2,3),a)` → `c(h(5,2),a)` that uses the rule `r` and equation `sum`, the term slice `c(f(2,` $\bullet_1$ `),` $\bullet_2$ `)` of `c(f(2,3),a)`, the term slice `c(h(5,2),` $\bullet_2$ `)` of `c(h(5,2),a)`, and the internal execution trace $\mathcal{T}_{int}$ = `g(2,3)` → `2 + 3` → `5` for the evaluation of the matching condition `Z` := `g(X, Y)`, which instantiates the extra-variable `Z` to the value `5`.

Intuitively, the computed value `5` in $\mathcal{T}_{int}$ descends from the input values `2` and `3` given to the variable arguments `X` and `Y` of the functions `g` and `f` (a more formal account of such a dependency computation is presented in Section 4.5). Thus, we conclude that the value `5` is a descendant (by the extra-variables relation) of the observed value `2` of the initial term slice `c(f(2,` $\bullet_1$ `),` $\bullet_2$ `)` of `c(f(2,3),a)`.

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

Part I

Backward Trace Analysis

Chapter 1

Backward Trace Slicing for Conditional Rewrite Theories

The backward tracing approach developed in this first part of the thesis aims to improve program analysis, comprehension and debugging of Maude programs by helping the user to *think backwards* —i.e., to deduce the conditions under which a program produces some observed output. In more conventional programming environments, for such an analysis, the programmer must repeatedly identify which statements in the code impact on the value of a given parameter at a given call, which is usually done manually without any assistance from the debugger. Our trace slicing technique tracks back reverse dependency and causality along execution traces and then cuts off irrelevant information that does not influence the data observed from the trace. In other words, when execution is stopped at a given computation point (typically, from the location where a fault is manifested), we are able to undo the effect of the last statement executed on the selected data by issuing the step-back facility provided by our slicer (for a given slicing criterion). Thus, by stepwisely reducing the amount of information to be inspected, it is easier for the user to locate errors because many computation steps (and the corresponding program statements involved in the step) can be ignored in the process of locating the program fault area. Moreover, during the trace slice computation, different types of information are computed that are related to the program execution, for example, contributing actions and data and noncontributing ones. After computation of a trace slice, all the noncontributing information is discarded from the trace, and we can even take advantage of the filtered information for the purpose of dynamic program slicing.

A backward trace slicing methodology for RWL was first proposed in [ABER11] that is only applicable to *unconditional* RWL theories, and, hence, it cannot be employed when the source program includes conditional equations and/or rules. The following example illustrates why

```
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 1.1: The _mod_ operator.

conditions cannot be disregarded by the slicing process.

Example 1.0.2

Consider the Maude specification in Figure 1.1, which computes the remainder of the division of two natural numbers, and the associated instrumented execution trace $\mathcal{T} = \texttt{4 mod 5} \rightarrow_K \texttt{4}$. Assume that we are interested in observing the origins of the target symbol 4 that appears in the final state. If we disregard the condition $\texttt{Y} > \texttt{X}$ of the first conditional equation, the slicing technique of [ABER11] computes the trace slice $\mathcal{T}^{\bullet} : \texttt{4 mod} \bullet \; \bullet\!\!\rightarrow \texttt{4}$, where there exist concrete instances of $\texttt{4 mod} \bullet$ that cannot be rewritten to 4 using the considered specification —e.g., $\texttt{4 mod 3} \not\rightarrow_K \texttt{4}$. By contrast, our novel conditional approach not only produces a trace slice, but also delivers a Boolean condition that establishes the valid instantiations of the input term that generate the observed data. In this specific case, our conditional slicing technique would deliver the pair $[\texttt{4 mod} \bullet \; \bullet\!\!\rightarrow \texttt{4}, \bullet > 4]$.

In this chapter, a slicing algorithm for conditional RWL computations is formulated. 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.

1.1 Backward Slicing for Execution Traces

In this section, we formulate a backward trace slicing algorithm that, given an instrumented execution trace $\mathcal{T} = s_0 \rightarrow_K^* s_n$ and a term slice $s_n^{\bullet}$

of s_n, generates the sliced counterpart $\mathcal{T}^\bullet = s_0^\bullet \bullet\!\!\to^* s_n^\bullet$ 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 instrumented trace slice.

Roughly speaking, given an instrumented execution trace $\mathcal{T} = s_0 \to_K^* s_n$, the instrumented trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is computed w.r.t. a *slicing criterion* —i.e., a user-defined term slice $s_n^\bullet$ of s_n that is generated by applying the *Tslice* function to s_n and a set of positions P in s_n the user wants to observe (in symbols, $s_n^\bullet = \mathit{Tslice}(s_n, P)$). Our technique inductively computes the association between the meaningful information of s_i and the meaningful information of s_{i-1}. For each such rewrite step, the conditions of the applied rule are recursively processed in order to ascertain the meaningful information of s_{i-1} from s_i, together with the accumulated compatibility condition $B_i^\bullet$. The technique proceeds backwards, from the final term s_n to the initial term s_0. A simplified, sliced instrumented trace is obtained where each s_i is replaced by the corresponding term slice $s_i^\bullet$.

Let us first introduce the notion of *backward trail*, which is a pair composed of a trace slice $\mathcal{T}^\bullet$ and a companion compatibility condition $B^\bullet$.

Definition 1.1.1 (Backward Trail) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and let $\mathcal{T} = s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1 \overset{r_2,\sigma_2,w_2}{\to_K} \cdots \overset{r_n,\sigma_n,w_n}{\to_K} s_n$ be an instrumented execution trace for s_0 in $\mathcal{R}$. Let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. A backward trail is a pair $[\mathcal{T}^\bullet, B^\bullet]$ where the first component is an instrumented trace slice $\mathcal{T}^\bullet = s_0^\bullet \bullet\!\!\to s_1^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $s_n^\bullet$, and $B^\bullet$ is a Boolean condition.*

Let us formalize now a calculus that allows the generation of backward trails w.r.t. a slicing criterion by means of a transition system $(\mathit{Conf}, \hookleftarrow)$ [Plo04] where *Conf* is a set of *configurations* and $\hookleftarrow$ is the transition relation that implements the backward trace slicing algorithm. Configurations are formally defined as follows.

Definition 1.1.2 *A configuration, written as $\langle \mathcal{T}, S^\bullet, B^\bullet \rangle$, consists of three components:*

– *the instrumented execution trace $\mathcal{T} = s_0 \to_K^* s_{i-1} \to_K s_i$ to be sliced;*

– *the term slice* $S^\bullet$*, that records the computed term slice* $s_i^\bullet$ *of* s_i
– *a Boolean condition* $B^\bullet$*.*

The transition system $(Conf, \looparrowright)$ is defined as follows.

Definition 1.1.3 *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a conditional rewrite theory, let* $\mathcal{T} = U \rightarrow_K^+ W$ *be an instrumented execution trace in* $\mathcal{R}$*, and let* $V \rightarrow_K W$ *be the last rewrite step of* $\mathcal{T}$*. Let* $B_W^\bullet$ *be a Boolean condition, and let* $W^\bullet$ *be a term slice of* W*. Given a set* $Conf$ *of configurations, the transition relation* $\looparrowright \subseteq Conf \times Conf$ *is the smallest relation that satisfies the following rule:*

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

Roughly speaking, the relation $\looparrowright$ transforms a configuration $\langle U \rightarrow_K^* V \rightarrow_K W, W^\bullet, B_W^\bullet \rangle$ into a configuration $\langle U \rightarrow_K^* V, V^\bullet, B_V^\bullet \rangle$ by calling the function $\mathit{slice\text{-}step}(V \rightarrow_K W, W^\bullet, B_W^\bullet)$ of Section 1.2, which returns a pair composed of a suitable term slice $V^\bullet$ of V and a Boolean condition $B_V^\bullet$ that updates the compatibility condition specified by $B_W^\bullet$. In other words, *slice-step* computes the rewrite step slice $V^\bullet \bullet\!\!\rightarrow W^\bullet$ of $V \rightarrow_K W$ updating the accumulated compatibility condition.

By using the transition system of Definition 1.1.3, the initial configuration $\langle s_0 \rightarrow_K^* s_n, s_n^\bullet, \mathit{true} \rangle$ is transformed until a terminal configuration $\langle s_0, s_0^\bullet, B_0^\bullet \rangle$ is reached. Then, the computed instrumented trace slice $\mathcal{T}^\bullet$ w.r.t. $s_n^\bullet$ is obtained by replacing each term s_i by the corresponding term slice $s_i^\bullet$, $i = 0, \ldots, n$, in the original instrumented execution trace $s_0 \rightarrow_K^* s_n$. The algorithm additionally returns, within the backward trail, the accumulated compatibility condition $B_0^\bullet$ contained in the terminal configuration.

More formally, the backward trail w.r.t. a slicing criterion is computed by the function *backward-slicing* as follows.

Definition 1.1.4 (Backward Trail) *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a conditional rewrite theory, and let* $\mathcal{T} = s_0 \rightarrow_K^* s_n$ *be an instrumented execution trace in* $\mathcal{R}$*. Let* $s_n^\bullet$ *be a slicing criterion for* $\mathcal{T}$*. Then, the* backward trail *that contains the instrumented trace slice* $\mathcal{T}^\bullet$ *of* $\mathcal{T}$ *w.r.t.* $s_n^\bullet$ *and the compatibility condition* $B_0^\bullet$*, is computed as follows:*

$$\mathit{backward\text{-}slicing}(s_0 \rightarrow_K^* s_n, s_n^\bullet) = [s_0^\bullet \bullet\!\!\rightarrow^* s_n^\bullet, B_0^\bullet]$$

iff there exists a transition sequence in $(Conf, \looparrowright)$

$$\langle s_0 \to_K^* s_n, s_n^\bullet, true\rangle \looparrowright^* \langle s_0, s_0^\bullet, B_0^\bullet\rangle$$

The following definitions provide a notion of correctness for a backward trail that will be used to prove the correction of the backward trace slicing algorithm described in this chapter.

First, let us show how we particularize a term slice by instantiating $\bullet$-variables with data that satisfy a given compatibility condition. A $B^\bullet$-compatible term slice concretization is formally defined as follows.

Definition 1.1.5 ($B^\bullet$-compatible 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 1.1.6

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.

Now, our notion of correctness is as follows.

Definition 1.1.7 (Correct Backward Trail) *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a conditional rewrite theory, and let* $\mathcal{T} = s_0 \overset{r_1,\sigma_1,w_1}{\to_K} s_1 \overset{r_2,\sigma_2,w_2}{\to_K} \cdots \overset{r_n,\sigma_n,w_n}{\to_K} s_n$ *be an instrumented execution trace for* s_0 *in* $\mathcal{R}$*. Let* $s_n^\bullet$ *be a slicing criterion for* $\mathcal{T}$*. A backward trail* $[\mathcal{T}^\bullet, B^\bullet]$*, with* $\mathcal{T}^\bullet = s_0^\bullet \bullet\!\!\to s_1^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$*, of* $\mathcal{T}$ *w.r.t.* $s_n^\bullet$ *is* correct *iff there exists an instrumented trace concretization* $\mathcal{T}' = s'_0 \to_K s'_1 \to_K \cdots \to_K s'_n$ *of* $\mathcal{T}^\bullet$ *with* $s_0^\bullet \propto^{B^\bullet} s'_0$*.*

Definition 1.1.7 provides an existential condition on backward trails. Roughly speaking, it ensures that, for *some* concrete instrumented trace $\mathcal{T}'$, whose first state s'_0 is a concretization of $s_0^\bullet$ compatible with $B^\bullet$, the very same rules involved in the sliced steps of $\mathcal{T}^\bullet$ can be applied again, at the same positions, in $\mathcal{T}'$ and each $s_i^\bullet$ is a term slice of s'_i.

```
mod BANK is
  pr INT .

  sorts Account Msg State Id .
  subsorts Account Msg < State .

  ops A B C D : -> Id [ctor] .
  op ac : Id Int -> Account [ctor] .
  ops credit debit : Id Int -> Msg [ctor] .
  op transfer : Id Id Int -> Msg  [ctor] .
  op empty-state : -> State [ctor] .
  op  _;_ : State State -> State [assoc comm id: empty-state] .

  vars Id Id1 Id2 : Id .
  vars b b1 b2 nb nb1 nb2 M : Int .

  crl [credit] : ac(Id,b) ; credit(Id,M) => ac(Id,nb) if nb := b + M .
  crl [debit] : ac(Id,b) ; debit(Id,M) => ac(Id,nb) if b >= M /\ nb := b - M .
  crl [transfer] : ac(Id1,b1) ; ac(Id2,b2) ; transfer(Id1, Id2,M)
                 => ac(Id1,nb1) ; ac(Id2,nb2)
                    if debit(Id1,M) ; ac(Id1,b1) => ac(Id1,nb1)
                       /\ credit(Id2,M) ; ac(Id2,b2) => ac(Id2,nb2) .
endm
```

Figure 1.2: Maude specification of a distributed banking system.

Example 1.1.8

Consider the Maude system module of Figure 1.2 that encodes a conditional rewrite theory modeling a simple, distributed banking system.

Each state of the system is modeled as a multiset (i.e., an associative and commutative list) of elements of the form $\mathtt{e}_1; \mathtt{e}_2; \ldots ; \mathtt{e}_\mathtt{n}$. Each element $\mathtt{e}_\mathtt{i}$ is either (i) a bank account `ac(Id,b)`, where `ac` is a constructor symbol (denoted by the Maude `ctor` attribute), `Id` is the owner of the account and `b` is the account balance; or (ii) a message modeling a debit, credit, or transfer operation. These account operations are implemented via three rewrite rules: namely, the `debit`, `credit`, and `transfer` rules. Consider the following instrumented execution trace:

$$
\begin{aligned}
\mathcal{T} = \;& \mathtt{ac(A,30)\ ;\ debit(A,5)\ ;\ credit(A,3)} \stackrel{\mathtt{debit}}{\to}_K \\
& \mathtt{ac(A,25)\ ;\ credit(B,3)} \stackrel{\mathtt{credit}}{\to}_K \\
& \mathtt{ac(A,28)}
\end{aligned}
$$

function $\textit{slice-step}(s \stackrel{r,\sigma,w}{\to}_K t, t^\bullet, B^\bullet_{prev})$ /* *Assuming*: $[r] : \lambda \Rightarrow \rho$ if C
1. **if** $w \notin \mathcal{MP}os(t^\bullet)$ **then** and $C = c_1 \wedge \ldots \wedge c_n$ */
2. $B^\bullet = B^\bullet_{prev}$
3. $s^\bullet = t^\bullet[\mathit{fresh}^\bullet]_{w'}$ with $w' \leq w \wedge t^\bullet_{|w'} = \bullet_i$, for some i
4. **else**
5. $\theta = \{x/\mathit{fresh}^\bullet \mid x \in \mathit{Dom}(\sigma)\}$
6. $\psi_{n+1} = (\theta mgu(\rho\theta, (t^\bullet_{|w})))_{\upharpoonright Dom(\sigma)}$
7. **for** $i = n$ **downto** 1 **do**
8. $(\psi_i, B^\bullet_i) = \textit{process-condition}(c_i, \sigma, \psi_{i+1})$
9. **od**
10. $B^\bullet = B^\bullet_{prev} \wedge (B^\bullet_1 \wedge \ldots \wedge B^\bullet_n)\psi_1$
11. $s^\bullet = t^\bullet[\lambda\psi_1]_w$
12. **fi**
13. **return** $(s^\bullet, B^\bullet)$

endf

Figure 1.3: Backward step slicing function.

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

$$\begin{array}{l}\texttt{ac(A,}\bullet_8\texttt{) ; debit(A, }\bullet_9\texttt{) ; credit(A, }\bullet_4\texttt{)} \stackrel{\texttt{debit}}{\bullet\!\!\to} \\ \texttt{ac(A, }\bullet_3\texttt{) ; credit(A, }\bullet_4\texttt{)} \stackrel{\texttt{credit}}{\bullet\!\!\to} \\ \texttt{ac(A, }\bullet_1\texttt{)}\end{array}$$

Then, $[\mathcal{T}^\bullet, \bullet_8 \geq \bullet_9]$ is a correct backward trail stating that the balance value (abstracted by $\bullet_8$) should exceed or be equal to the withdrawal value abstracted by $\bullet_9$.

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

1.2 The Function *slice-step*

The function *slice-step*, which is outlined in Figure 1.3, takes three parameters as input: a rewrite step $\mu : s \stackrel{r,\sigma,w}{\to}_K t$ with $[r] : \lambda \Rightarrow \rho$ if C, a term

slice $t^\bullet$ of t, and a compatibility condition $B^\bullet_{prev}$. It delivers as outcome the term slice $s^\bullet$ of s and an updated compatibility condition $B^\bullet$.

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 $t^\bullet$. Actually, the rewriting position w might not even occur in $t^\bullet$, hence we consider the prefix w' of w that points to a $\bullet$-variable in $t^\bullet$, i.e., $t^\bullet_{|w'}$ is a $\bullet$-variable. This position exists and is unique. Now, since no new relevant information descends from the term slice $t^\bullet$, *slice-step* returns a variant $t^\bullet[fresh^\bullet]_{w'}$ of $t^\bullet$ where $t^\bullet_{|w'}$ has been replaced by a new fresh $\bullet$-variable that completely abstracts the redex computed by μ.

Example 1.2.1

Consider the Maude specification of Example 1.1.8 and the following rewrite step μ: `ac(A,30) ; debit(A,5) ; credit(A,3)` $\stackrel{\texttt{debit}}{\rightarrow}_K$ `ac(A,25) ; credit(A,3)`. Let $\bullet_1$ `; credit(A,3)` be a term slice of `ac(A,25) ; credit(A,3)`. Since the rewrite step μ occurs at position $1 \notin \mathcal{MP}os(\bullet_1$ `; credit(A,3)`$)$, the term `ac(A,25)` introduced by μ in `ac(A,25) ; credit(A,3)` is completely ignored in $\bullet_1$ `; credit(A,3)`. Hence, the computed term slice for `ac(A,30) ; debit(A,5) ; credit(A,3)` is $\bullet_2$ `; credit(A,3)`, where $\bullet_2$ is a fresh variable generated by the function *fresh*$^\bullet$.

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 conditional rewrite step, which is based on an inductive process that is obtained by recursively processing the conditions of the applied rule. More specifically, we initially define the substitution $\theta = \{x/fresh^\bullet \mid x \in Dom(\sigma)\}$ that binds each variable in the domain of σ 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, the algorithm computes a sequence of substitutions that incrementally refine θ by using the following two-step procedure.

In the first phase, the relevant information contained in the term slice $t^\bullet_{|w}$ of the contractum $t_{|w}$ is retrieved, while in the second phase, relevant symbols that come from evaluating the rule condition are recognized.

function *process-condition*(c, σ, ψ)
1. **case** c **of**
2. $(p := m) \vee (m \Rightarrow p)$ **:** /* matching conditions
3. **if** $(m\sigma = p\sigma)$ and rewrite expressions */
4. $\delta = mgu(m, p\psi)$
5. **else**
6. $[(m\sigma)^\bullet \ \to^+ (p\sigma)^\bullet, B^\bullet] = backward\text{-}slicing(m\sigma \to_K^+ p\sigma, p\psi)$
7. $\delta = mgu(m, (m\sigma)^\bullet)$
8. **fi**
9. **return** $((\delta \Uparrow \psi)_{\lceil Dom(\psi)}, B^\bullet)$
10. e **:** /* equational conditions */
11. **return** (ψ, e)
12. **end case**

endf

Figure 1.4: Condition processing function.

Phase 1. We compute the substitution composition ψ_{n+1} (restricted to $Dom(\sigma)$'s variables) of θ and the most general unifier between the sliced contractum $t^\bullet_{|w}$ and the right-hand side ρ of the applied rule $[r] : \lambda \Rightarrow \rho$ if C instantiated by θ. This allows us to catch (and store in ψ_{n+1} the meaningful information of the sliced contractum $t^\bullet_{|w}$ while those data that do not appear at meaningful positions are ignored.

Note that the use of unification within our slicing methodology somehow resembles the unification-based, parameter-passing mechanism that is used in narrowing [Fay79, Sla74].

Example 1.2.2

Consider the rewrite theory in Example 1.1.8 together with the following rewrite step $\mu_{\texttt{debit}}$: `ac(A,30) ; debit(A,5)` $\overset{\texttt{debit}}{\to}_K$ `ac(A,25)` that involves the application (at position $w = \Lambda$) of the `debit` rule whose right-hand side is $\rho_{\texttt{debit}}$ = `ac(Id,nb)`. Let $t^\bullet$ = `ac(A,`$\bullet_1$`)` be a term slice of `ac(A,25)` and $B^\bullet_{prev} = true$. Then, the initially ascertained substitution for $\mu_{\texttt{debit}}$ is

$$\theta = \{\texttt{Id}/\bullet_2, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_5\}$$

and we compute the substitution ψ_{n+1} as follows.

$$\begin{aligned}\psi_{n+1} &= (\theta mgu(\texttt{ac(}\bullet_2\texttt{, }\bullet_5\texttt{), ac(A, }\bullet_1\texttt{)}))_{\restriction\{\texttt{Id,b,M,nb}\}} \\ &= (\theta\{\bullet_2/\texttt{A}, \bullet_5/\bullet_1\})_{\restriction\{\texttt{Id,b,M,nb}\}} \\ &= (\{\texttt{Id}/\bullet_2, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_5\}\{\bullet_2/\texttt{A}, \bullet_5/\bullet_1\})_{\restriction\{\texttt{Id,b,M,nb}\}} \\ &= \{\texttt{Id}/\texttt{A}, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_1\}\end{aligned}$$

Note that by replacing the uninformed binding $\texttt{Id}/\bullet_2$, with $\texttt{Id}/\texttt{A}$ in ψ_{n+1} we catch the meaningful value `A` for the variable `Id` of the right-hand side of the rule `debit`.

Phase 2. Let $C\sigma = c_1\sigma \wedge \ldots \wedge c_n\sigma$ be the instance of the condition in the rule r that enables the rewrite step μ. We process each (sub)condition $c_i\sigma$, $i = 1, \ldots, n$, in reversed evaluation order, i.e., from $c_n\sigma$ to $c_1\sigma$, by using the auxiliary function *process-condition* given in Figure 1.4 that generates a pair $(\psi_i, B_i^\bullet)$ such that ψ_i is used to further refine the partially ascertained substitution ψ_{i+1} that is computed by incrementally analyzing conditions $c_n\sigma, \ldots, c_{i+1}\sigma$, and $B_i^\bullet$ is a Boolean condition that is derived from the analysis of the condition c_i.

The processing of the whole $C\sigma$ yields a substitution ψ_1, which includes the relevant instantiations of the considered conditional rewrite step μ w.r.t. $t^\bullet_{|w}$.

Now, the term slice $t^\bullet$ is obtained from $s^\bullet$ by replacing its subterm at position w with the instance $(\lambda\psi_1)$ of the left-hand side of the applied rule r. This way, all the relevant/irrelevant information identified is transferred into the resulting sliced term $s^\bullet$. Furthermore, $B^\bullet$ is built by collecting all the Boolean compatibility conditions $B_i^\bullet$ delivered by *process-condition* and instantiating them with ψ_1.

It is worth noting that *process-condition* handles matching conditions, rewrite expressions and equational conditions differently. More specifically, the pair (ψ_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

instrumented step $\mu : s \stackrel{r,\sigma,w}{\rightarrow}_K t$, recall that c is evaluated as follows: first, $m\sigma$ is reduced to its canonical form $m\sigma\downarrow_{\Delta,B}$, and then the condition $m\sigma\downarrow_{\Delta}=_B p\sigma$ is checked. In our framework, this corresponds to producing an internal instrumented execution trace that transforms $m\sigma$ into $p\sigma$.

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

Case i. If $p\sigma = m\sigma$, there is no need to generate the canonical form of $m\sigma$, since $p\sigma$ and $m\sigma$ are the same term. Hence, we discover new (possibly) relevant bindings for variables in m by computing the mgu δ between m and $p\psi$. Then, the algorithm returns the parallel composition of δ and ψ (restricted to $Dom(\psi)$'s variables) that updates the input substitution ψ with the new bindings encoded in δ.

Since the co-domains of ψ and δ contain no variables apart from $\bullet$-symbols, the parallel composition is harmless for any other bindings different from $x/\bullet$.

Example 1.2.3

Consider the following substitutions $\delta = \{X/h(2), Y/4\}$ and $\psi = \{X/h(\bullet), Z/5\}$. Then, the parallel composition of δ and ψ is as follows:

$$\begin{aligned}\delta \Uparrow \psi = \;& mgu(f(X,Y,X,Z), f(h(2),4,h(\bullet),5)) \\ = \;& \{X/h(2), Y/4, Z/5, \bullet/2\}\end{aligned}$$

Case ii. When $p\sigma \neq m\sigma$, the analysis of the matching condition $p := m$ during the slicing process of μ implies slicing the (internal) instrumented execution trace $\mathcal{T}_{int} = m\sigma \rightarrow_K^+ p\sigma$, which is done by recursively invoking the function *backward-slicing* for execution trace slicing with respect to the slicing criterion given by the instantiation of p with ψ, where ψ is the substitution that records the meaningful information computed so far. That is, $[(m\sigma)^\bullet \bullet\!\!\rightarrow^+ (p\sigma)^\bullet, B^\bullet] = backward\text{-}slicing(m\sigma \rightarrow_K^+$

$p\sigma, p\psi$). The result delivered by the function *backward-slicing* is a backward trail with the instrumented trace slice $(m\sigma)^\bullet \bullet\!\!\to^+ (p\sigma)^\bullet$ and compatibility condition $B^\bullet$, from which new relevant bindings for m's variables can be derived. This is done by computing the mgu δ between m and $(m\sigma)^\bullet$; then, the parallel composition of δ and ψ (restricted to $Dom(\psi)$'s variables) is computed in order to reconciliate ψ with the new bindings in δ, and the composed substitution is delivered together with the compatibility condition $B^\bullet$.

Example 1.2.4

Consider the rewrite step $\mu_{\texttt{debit}}$ of Example 1.2.2 together with the refined substitution $\psi_{n+1} = \{\texttt{Id/A}, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_5\}$. We process the condition `nb := b - M` of `debit` rule by considering an internal instrumented execution trace $\mathcal{T}_{int} = \texttt{30 - 5} \to_K \texttt{25}$. By invoking the *backward-slicing* function with the slicing criterion given by $\texttt{nb}\psi_{n+1} = \bullet_5$, the resulting backward trail is $[\bullet_5 \;\bullet\!\!\to\; \bullet_5, true]$. Then, we compute $\delta = mgu(\texttt{b - M}, \bullet_5) = \{\bullet_5/\texttt{b - M}\}$ and the final outcome is $(\delta \Uparrow \psi_{n+1})_{\restriction\{\texttt{Id,b,M,nb}\}} = \{\texttt{Id/A}, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_3 \texttt{ - } \bullet_4, \bullet_5/\bullet_3 \texttt{ - } \bullet_4\}_{\restriction\{\texttt{Id,b,M,nb}\}} = \{\texttt{Id/A}, \texttt{b}/\bullet_3, \texttt{M}/\bullet_4, \texttt{nb}/\bullet_3 \texttt{ - } \bullet_4\}$.

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

– **Equational conditions.** During the execution of the rewrite step $\mu : s \overset{r,\sigma,w}{\to}_K t$, 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 new relevant instantiations can be identified for variables that appear in e. However, the equational condition e must be recorded within the compatibility condition $B^\bullet$ for the considered conditional rewrite step. In other words, after processing an equational condition e, we deliver the tuple (ψ, e), with ψ the unaltered, input substitution

Example 1.2.5

Consider the refined substitution given in Example 1.2.4

$$\psi_{n+1} = \{\texttt{Id/A}, \texttt{b/}\bullet_3, \texttt{M/}\bullet_4, \texttt{nb/}\bullet_5\}$$

together with the rewrite step $\mu_{\texttt{debit}}$ of Example 1.2.2 that involves the application of the `debit` rule. After processing the condition `b >= M` of `debit`, we deliver $B^\bullet = \texttt{b >= M}$

In the following section, we provide a notion of correctness that holds for each backward trail computed by the backward trace slicing technique described in this chapter.

1.3 Correctness of Backward Trace Slicing

The correctness of backward trace slicing is formally defined as follows.

Proposition 1.3.1 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K \cdots \stackrel{r_n,\sigma_n,w_n}{\rightarrow}_K s_n$ be an instrumented execution trace for s_0 in $\mathcal{R}$, with $n \geq 0$, and let $s_n^\bullet$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet, B_0^\bullet]$ computed by backward-slicing$(\mathcal{T}, s_n^\bullet)$ is a backward trail.*

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

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

iff there exists a transition sequence $\mathcal{S}$ in $(\textit{Conf}, \looparrowright)$

$$\langle s_0 \rightarrow_K^* s_n, s_n^\bullet, \textit{true}\rangle \looparrowright \langle s_0 \rightarrow_K^* s_{n-1}, s_{n-1}^\bullet, B_{n-1}^\bullet\rangle \looparrowright^* \langle s_0, s_0^\bullet, B_0^\bullet\rangle \qquad (1.1)$$

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

$$B_i^\bullet = \textit{true} \;\wedge\; \bigwedge_{j=0}^{n-1} b_j \Phi_j \qquad \text{for all } i = 0, \ldots, n-1 \qquad (1.2)$$

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

Now, to prove the proposition, simply observe that

1. each $s^\bullet_i$, $i = 0, \dots, n-1$, is built in the transition

$$\langle s_0 \to^*_K s_{i+1}, s^\bullet_{i+1}, B^\bullet_{i+1}\rangle \looparrowright \langle s_0 \to^*_K s_i, s^\bullet_i, B^\bullet_i\rangle$$

   by calling the function *slice-step* on the rewrite step $s_i \stackrel{r_{i+1},\sigma_{i+1},w_{i+1}}{\to_K} s_{i+1}$, the term slice $s^\bullet_{i+1}$, and the Boolean condition $B^\bullet_{i+1}$. Specifically, the call to *slice-step* generates a term $s^\bullet_i \in \tau(\Sigma, \mathcal{V}^\bullet)$ that is derived from $s^\bullet_{i+1}$ by either replacing an appropriate subterm of $s^\bullet_{i+1}$ with a fresh $\bullet$-variable generated by invoking $fresh^\bullet$ (line 3 of the function *slice-step*) or replacing an appropriate subterm of $s^\bullet_{i+1}$ with an instance of the left-hand side of the rule r_{i+1} (line 11 of the function *slice-step*). In both cases, $s^\bullet_i \in \tau(\Sigma, \mathcal{V}^\bullet)$.

2. By 1.2, $B^\bullet_0$ is the conjunction of Boolean conditions $true \wedge b_{n-1}\Phi_{n-1} \wedge \dots \wedge b_0\Phi_0$. Hence, $B^\bullet_0$ is a Boolean condition.

Therefore, $[s^\bullet_0 \bullet\!\!\to^* s^\bullet_n, B^\bullet_0]$ is a backward trail by Definition 1.1.1. ■

Theorem 1.3.2 (Correctness of Backward Trace Slicing) *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = s_0 \stackrel{r_1,\sigma_1,w_1}{\to_K} \cdots \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n$ be an instrumented execution trace for s_0 in $\mathcal{R}$, with $n \geq 0$, and let $s^\bullet_n$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s^\bullet_0 \bullet\!\!\to \cdots \bullet\!\!\to s^\bullet_n, B^\bullet_0]$ computed by backward-slicing$(\mathcal{T}, s^\bullet_n)$ is a correct backward trail of $\mathcal{T}$ w.r.t $s^\bullet_n$.*

Proof. Let $\mathcal{T} = s_0 \stackrel{r_1,\sigma_1,w_1}{\to_K} \cdots \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n$ be an instrumented execution trace in the conditional rewrite theory $\mathcal{R}$, with $n \geq 0$, and let $s^\bullet_n$ be a slicing criterion for $\mathcal{T}$. Then,

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

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

$$\langle s_0 \to^*_K s_n, s^\bullet_n, true\rangle \looparrowright \langle s_0 \to^*_K s_{n-1}, s^\bullet_{n-1}, B^\bullet_{n-1}\rangle \looparrowright^* \langle s_0, s^\bullet_0, B^\bullet_0\rangle \quad (1.3)$$

By Proposition 1.3.1, the outcome $[s_0^\bullet \;\bullet\!\!\rightarrow^*\; s_n^\bullet, B_0^\bullet]$ is a backward trail. Therefore, we just need to prove that $[s_0^\bullet \bullet\!\!\rightarrow^* s_n^\bullet, B_0^\bullet]$ is correct, that is, there exists a term s'_0 with $s_0^\bullet \propto^{B_0^\bullet} s'_0$, and an instrumented trace concretization $s'_0 \to_K s'_1 \to_K \cdots \to_K s'_n$ of $[s_0^\bullet \;\bullet\!\!\rightarrow^*\; s_n^\bullet, B_0^\bullet]$ such that $s'_i \to_K s'_{i+1}$ is the rewrite step $s'_i \stackrel{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to_K} s'_{i+1}$. , $i = 0, \ldots, n-1$.

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

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

$\mathcal{N}(\mathcal{T}) > \mathbf{0}$. We have a nonempty instrumented execution trace $\mathcal{T} = s_0 \stackrel{r_1,\sigma_1,w_1}{\to_K} \cdots \stackrel{r_{n-1},\sigma_{n-1},w_{n-1}}{\to_K} s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n$ and a slicing criterion $s_n^\bullet$ to which backward trace slicing is applied. Specifically, by applying *backward-slicing*$(\mathcal{T}, s_n^\bullet)$, the transition sequence $\mathcal{S}$ in 1.3 is produced.

We consider the instrumented execution trace $\mathcal{T}^{n-1} = s_0 \stackrel{r_1,\sigma_1,w_1}{\to_K} \cdots \stackrel{r_{n-1},\sigma_{n-1},w_{n-1}}{\to_K} s_{n-1}$. Since $\mathcal{N}(\mathcal{T}^{n-1}) < \mathcal{N}(\mathcal{T})$, the inductive hypothesis holds for $\mathcal{T}^{n-1}$ and the slicing criterion $s_{n-1}^\bullet$. Thus, *backward-slicing*$(s_0 \to_K^* s_{n-1}, s_{n-1}^\bullet) = [s_0^\bullet \bullet\!\!\rightarrow^* s_{n-1}^\bullet, B_0^{\bullet'}]$ and $[s_0^\bullet \bullet\!\!\rightarrow^* s_{n-1}^\bullet, B_0^{\bullet'}]$ is a correct backward trail.

Then, we consider the rewrite step $s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n$. By Definition 1.1.3, the sliced counterpart, $s_{n-1}^\bullet \bullet\!\!\rightarrow s_n^\bullet$, of the rewrite step, is computed in the first transition of $\mathcal{S}$ (that is, $\langle s_0 \to_K^* s_n, s_n^\bullet, \mathit{true}\rangle \hookleftarrow \langle s_0 \to_K^* s_{n-1}, s_{n-1}^\bullet, B_{n-1}^\bullet\rangle$) by executing *slice-step*$(s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n, s_n^\bullet, \mathit{true})$ which returns the pair $(s_{n-1}^\bullet, B_{n-1}^\bullet)$. We distinguish the two following cases.

Case $w_n \notin \mathcal{MP}os(s_n^\bullet)$**.** In this case, *slice-step*$(s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n, s_n^\bullet, \mathit{true})$ yields the pair $(s_{n-1}^\bullet, B_{n-1}^\bullet)$ such that $s_{n-1}^\bullet = s_n^\bullet[\bullet_f]_{w'}$ where $w' \leq w$, $s^\bullet_{n|w'}$ is a $\bullet$-variable and $\bullet_f$ is a fresh $\bullet$-variable that has been generated by invoking *fresh*$^\bullet$, and $B_{n-1}^\bullet = \mathit{true}$. Thus, the backward trail must be of the form $[s_0^\bullet \bullet\!\!\rightarrow^* s_n^\bullet[\bullet_f]_{w'} \bullet\!\!\rightarrow s_n^\bullet, B_0^{\bullet'} \;\wedge\; \mathit{true}]$, where $s_n^\bullet[\bullet_f]_{w'}$ is just a renaming of $s_n^\bullet$.

Since $[s_0^\bullet \bullet\!\!\to^* s_n^\bullet[\bullet_f]_{w'}, B_0^{\bullet'}]$ is a correct backward trail, there exists a term s'_0 with $s_0^\bullet \propto^{B_0^{\bullet'}} s'_0$, and an instrumented trace concretization $s'_0 \to_K s'_1 \to_K \cdots \to_K s'_{n-1}$ of $s_0^\bullet \bullet\!\!\to^* s_n^\bullet[\bullet_f]_{w'}$ such that $s'_i \to_K s'_{i+1}$ is the rewrite step $s'_i \stackrel{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to_K} s'_{i+1}$, $i = 0,\ldots,n-2$.

Then, it also holds that $s_0^\bullet \propto^{B_0^{\bullet'} \wedge\, true} s'_0$. Furthermore, for the instrumented execution trace $\mathcal{T}' = (s'_0 \stackrel{r_1,\sigma'_1,w_1}{\to_K} \cdots \stackrel{r_{n-1},\sigma_{n-1},w_{n-1}}{\to_K} s'_{n-1} \stackrel{r_n,\sigma'_n,w_n}{\to_K} s'_{n-1}[\rho_n\sigma'_n]_{w_n} = s'_n)$ in $\mathcal{R}$, where ρ_n is the right-hand side of r_n, we have:

i) $s'_i \to_K s'_{i+1}$ is the rewrite step $s'_i \stackrel{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to_K} s'_{i+1}$, $i = 0,\ldots,n-1$;
ii) $s_i^\bullet$ is a term slice of s'_i, for all $i = 0,\ldots,n$ (that is, $\mathcal{T}'$ is an instrumented trace concretization of $s_0^\bullet \bullet\!\!\to^* s_{n-1}^\bullet[\bullet_f]_{w'} \bullet\!\!\to s_n^\bullet$), with $s_{n-1}^\bullet = s_n^\bullet[\bullet_f]_{w'}$.

Therefore, $[s_0^\bullet \bullet\!\!\to^* s_{n-1}^\bullet \bullet\!\!\to s_n^\bullet, B_0^{\bullet'} \wedge\ true]$ is a correct backward trail.

Case $w_n \in \mathcal{MP}os(s_n^\bullet)$**.** Let r_n be the rewrite rule $[r_n] : \lambda_n \Rightarrow \rho_n$ *if* C_n. In this case, *slice-step*$(s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n, s_n^\bullet, true)$ yields the pair $(s_{n-1}^\bullet, B_{n-1}^\bullet)$ such that $s_{n-1}^\bullet = s_n^\bullet[\lambda_n\psi]_{w_n}$, for some substitution ψ, and $B_{n-1}^\bullet$ is the conjunction of some equational conditions needed to evaluate the condition $C_n\sigma_n$ in the rewrite step $s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n$. Thus, the backward trail must be of the form $[s_0^\bullet \bullet\!\!\to^* s_n^\bullet[\lambda\psi]_{w_n} \bullet\!\!\to s_n^\bullet, B_0^{\bullet'} \wedge B_{n-1}^\bullet]$, where $B_0^{\bullet'}$ is the compatibility condition computed by slicing the instrumented execution trace $\mathcal{T}^{n-1}$.

Now, we consider the backward trail $[s_0^\bullet \bullet\!\!\to^* s_{n-1}^\bullet, B_0^{\bullet'}]$, which is correct by inductive hypothesis. By Definition 1.1.7, we know that there exists a term s'_0 with $s_0^\bullet \propto^{B_0^{\bullet'}} s'_0$, and an instrumented trace concretization $s'_0 \to s'_1 \to_K \cdots \to_K s'_{n-1}$ of $s_0^\bullet \bullet\!\!\to^* s_{n-1}^\bullet$ such that $s'_i \to_K s'_{i+1}$ is the rewrite step $s'_i \stackrel{r_{i+1},\sigma'_{i+1},w_{i+1}}{\to_K} s'_{i+1}$, $i = 0,\ldots,n-2$.

In particular, we can choose $s'_0 = s_0$. This way, all the descendants of s_0 will be kept in the instrumented execution trace

$s_0 = s'_0 \rightarrow_K s'_1 \rightarrow_K \cdots \rightarrow_K s'_{n-1}$. Hence, $s'_{n-1|w_n} = s_{n-1|w_n} = \lambda_n\sigma_n$. Furthermore, $s^\bullet_0 \propto^{B^{\bullet\prime}_0 \wedge B^\bullet_{n-1}} s_0 = s'_0$. Thus, by definition of rewriting, $s'_{n-1} \overset{r_n,\sigma'_n,w_n}{\rightarrow_K} s'_n$ with $s'_{n|w_n} = \rho_n\sigma'_n = \rho_n\sigma_n = s_{n|w_n}$. Also, $s^\bullet_n$ is a term slice of s'_n, since $s^\bullet_n$ is a slicing criterion for $\mathcal{T}$.

Therefore, $[s^\bullet_0 \bullet\!\!\rightarrow^* s^\bullet_n[\lambda\psi]_{w_n} \bullet\!\!\rightarrow s^\bullet_n, B^{\bullet\prime}_0 \wedge B^\bullet_{n-1}]$ is a backward trail correct.

■

Chapter 2

The *i*Julienne System

The backward trace slicing methodology for conditional RWL computations described in Chapter 1 has been implemented in the slicing-based trace analysis tool *i*Julienne, which is available at [iJu12]. The core engine of *i*Julienne is written in Maude and consists of about 250 Maude function definitions (approximately 1.5K lines of source code). The *i*Julienne tool is a stand-alone application that can be invoked as a Maude command or used online through a Java web service. It allows the analysis of general rewrite theories that may contain (conditional) rules and equations, built-in operators, and algebraic axioms. The implementation uses meta-level capabilities and relies on the very efficient Maude system [CDE+07]. Actually, the current version of the Maude interpreter can do more than 3 million rewrites per second on state-of-the-art processors, and the Maude compiler can reach up to 15 million rewrites per second. The *i*Julienne user interface is based on the AJAX technology, which allows the Maude back-end to be used through the WWW.

*i*Julienne generalizes and supersedes previous trace slicing tools such as the trace slicer for unconditional RWL theories described in [ABER11], and the conditional slicer Julienne presented in [ABFR12b].

The architecture of *i*Julienne, which is depicted in Figure 2.1, 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

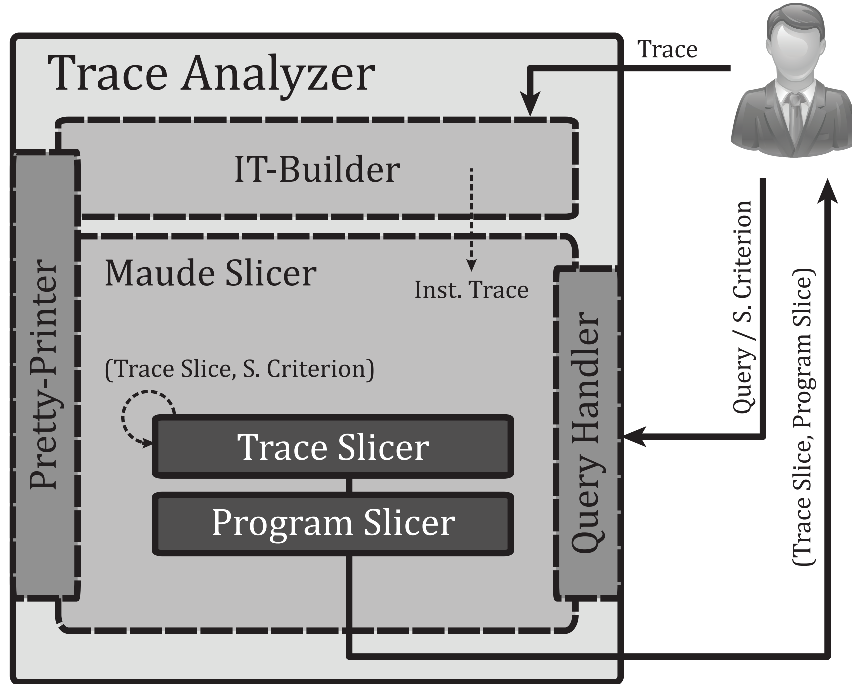


Figure 2.1: *i*Julienne architecture.

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

The way in which the slicing criteria are defined has been greatly improved in *i*Julienne. Besides supporting mouse click events that can

select any information piece in the state, a **Query handling** facility is included that allows huge execution traces to be queried by simply providing a filtering pattern (the query) that specifies a set of symbols to monitor and also selects those states that match the pattern. A pattern language with wild cards ? and _ is used to identify (resp. discard) the relevant (resp. irrelevant) data inside the states.

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

2.1 *i*JULIENNE at Work

In general, conventional debugging is an inefficient and time-consuming approach for understanding program behavior, especially when a programmer is interested in observing only those parts of the program execution that relate to the incorrect output. In order to make program debugging and comprehension more efficient, it is important to focus the programmer's attention on the essential components (actions, states, equations and rules) of the program and their execution. Backward trace slicing provides a means to achieve this by pruning away the unrelated pieces of the computation [Wei81].

2.1.1 Debugging Maude Programs with *i*JULIENNE

In debugging, one is often interested in analyzing a particular execution of a program that exhibits anomalous behavior. However, the execution of Maude programs typically generates large and clumsy traces that are hard to browse and understand even when the programmer is assisted by tracing tools such as the Maude built-in tracing facility. This is because the tracer does not provide any means for identifying the contributing program parts of the program being debugged, and does not allow the

```
mod BANK_ERR is
  pr INT .

  sorts Account Msg State Id .
  subsorts Account Msg < State .

  ops A B C D : -> Id [ctor] .
  op ac : Id Int -> Account [ctor] .
  ops credit debit : Id Int -> Msg [ctor] .
  op transfer : Id Id Int -> Msg  [ctor] .
  op empty-state : -> State [ctor] .
  op  _;_ : State State -> State [assoc comm id: empty-state] .

  vars Id Id1 Id2 : Id .
  vars b b1 b2 nb nb1 nb2 M : Int .

  crl [credit] : ac(Id,b) ; credit(Id,M) => ac(Id,nb) if nb := b + M .
  crl [debitERR] : ac(Id,b) ; debit(Id,M) => ac(Id,nb) if nb := b - M .
  crl [transfer] : ac(Id1,b1) ; ac(Id2,b2) ; transfer(Id1, Id2,M)
                  => ac(Id1,nb1) ; ac(Id2,nb2)
                     if debit(Id1,M) ; ac(Id1,b1) => ac(Id1,nb1)
                        /\ credit(Id2,M) ; ac(Id2,b2) => ac(Id2,nb2) .
endm
```

Figure 2.2: Faulty Maude specification of a distributed banking system.

programmer to distinguish related computations from unrelated computations. The inspection of these traces for debugging purposes is thus a cumbersome task that very often leads to no conclusion. In this scenario, backward trace slicing can play a meaningful role, since it can automatically reduce the size of the analyzed execution trace keeping track of all and only those symbols that impact on an error or anomaly in the trace.

Basically, the idea is to feed *i*Julienne with an execution trace $\mathcal{T}$ that represents a wrong behavior of a given Maude program, together with a slicing criterion that observes an erroneous outcome. The resulting trace slice is typically much smaller than the original one, since it only includes the information that is responsible for the production of the erroneous outcome. Thus, the programmer can easily navigate through the trace slice and repeatedly refine the slicing criteria for program bugs to hunt, as shown in the following examples.

Example 2.1.1

Consider the Maude program `BANK_ERR` of Figure 2.2, which is a faulty mutation of the distributed banking system specified in Figure 1.2. More precisely, the rule `debit` has been replaced by the rule `debitERR` in which

Trace information ×

State	Label	Trace	Trace Slice
1	'Start	ac(A,50) ; ac(B,20) ; ac(C,20) ; ac(D,20) ; credit(A,10) ; credit(D,40) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4)	ac(•,•) ; ac(•,•) ; ac(•,20) ; ac(•,•) ; credit(•,•) ; credit(•,•) ; debit(•,50) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4)
2	fromBnf	ac(B,20) ; ac(C,20) ; ac(D,20) ; credit(D,40) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,50) ; credit(A,10)	ac(•,•) ; ac(•,20) ; ac(•,•) ; credit(•,•) ; debit(•,50) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•) ; credit(•,•)
3	credit	ac(B,20) ; ac(C,20) ; ac(D,20) ; credit(D,40) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,60)	ac(•,•) ; ac(•,20) ; ac(•,•) ; credit(•,•) ; debit(•,50) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•)
4-5	toBnf-fromBnf	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,20) ; credit(D,40)	ac(•,•) ; ac(•,•) ; ac(•,20) ; debit(•,50) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•) ; credit(•,•)
6	credit	ac(A,60) ; ac(B,20) ; ac(C,20) ; debit(C,50) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,60)	ac(•,•) ; ac(•,•) ; ac(•,20) ; debit(•,50) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•)
7-8	toBnf-fromBnf	ac(A,60) ; ac(B,20) ; ac(D,60) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(C,20) ; debit(C,50)	ac(•,•) ; ac(•,•) ; ac(•,•) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,20) ; debit(•,50)
9	debitERR	ac(A,60) ; ac(B,20) ; ac(D,60) ; debit(D,5) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(C,- 30)	ac(•,•) ; ac(•,•) ; ac(•,•) ; debit(•,•) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,- 30)
10-11	toBnf-fromBnf	ac(A,60) ; ac(B,20) ; ac(C,- 30) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,60) ; debit(D,5)	ac(•,•) ; ac(•,•) ; ac(•,- 30) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•) ; debit(•,•)
12	debitERR	ac(A,60) ; ac(B,20) ; ac(C,- 30) ; transfer(A,C,15) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(D,55)	ac(•,•) ; ac(•,•) ; ac(•,- 30) ; transfer(•,•,15) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•)
13-14	toBnf-fromBnf	ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,60) ; ac(C,- 30) ; transfer(A,C,15)	ac(•,•) ; ac(•,•) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•) ; ac(•,- 30) ; transfer(•,•,15)
15	transfer	ac(B,20) ; ac(D,55) ; transfer(A,D,20) ; transfer(B,C,4) ; ac(A,45) ; ac(C,- 15)	ac(•,•) ; ac(•,•) ; transfer(•,•,•) ; transfer(•,•,4) ; ac(•,•) ; ac(•,- 15)
16-17	toBnf-fromBnf	ac(B,20) ; ac(C,- 15) ; transfer(B,C,4) ; ac(A,45) ; ac(D,55) ; transfer(A,D,20)	ac(•,•) ; ac(•,- 15) ; transfer(•,•,4) ; ac(•,•) ; ac(•,•) ; transfer(•,•,•)
18	transfer	ac(B,20) ; ac(C,- 15) ; transfer(B,C,4) ; ac(A,25) ; ac(D,75)	ac(•,•) ; ac(•,- 15) ; transfer(•,•,4) ; • ; •
19-20	toBnf-fromBnf	ac(A,25) ; ac(D,75) ; ac(B,20) ; ac(C,- 15) ; transfer(B,C,4)	• ; • ; ac(•,•) ; ac(•,- 15) ; transfer(•,•,4)
21	transfer	ac(A,25) ; ac(D,75) ; ac(B,16) ; ac(C,- 11)	• ; • ; • ; ac(•,- 11)
22	toBnf	ac(A,25) ; ac(B,16) ; ac(C,- 11) ; ac(D,75)	• ; • ; ac(•,- 11) ; •
Total size:		1689	295
Reduction Rate: 83%			

Figure 2.3: *i*JULIENNE output for the trace $\mathcal{T}_{bank}$ w.r.t. the slicing criterion `ac( • ,-11)`.

we have intentionally omitted the equational condition `b >= M`. Roughly speaking, the considered specification always authorizes withdrawals even in the erroneous case when the amount of money to be withdrawn is greater than the account balance.

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

```
ac(A,50) ; ac(B,20) ; ac(D,20) ; ac(C,20) ;
```

```
transfer(B,C,4) ; transfer(A,C,15) ;
debit(D,5) ; credit(A,10) ; transfer(A,D,20) ;
debit(C,50) ; credit(D,40)
```

and ends in the final state

```
ac(A,25) ; ac(B,16) ; ac(D,75) ; ac(C,-11)
```

We observe that the final state contains a negative balance for client `C`, which is a clear symptom of malfunction of the `BANK_ERR` specification, if we assume that balances must be non-negative numbers according to the semantics intended by the programmer. Therefore, we execute *i*JULIENNE on the trace $\mathcal{T}_{bank}$ w.r.t. the slicing criterion `ac(•,-11)` that observes the negative balance of the term `ac(C,-11)` in order to determine the cause of such a negative balance. The output delivered by *i*JULIENNE is given in Figure 2.3 and shows the (instrumented) input trace $\mathcal{T}_{bank}$ in the **Trace** column, the simplified trace $\mathcal{T}^{\bullet}_{bank}$ in the **Trace Slice** column, and some other auxiliary data such as the size of the trace and the corresponding trace slice, the reduction rate achieved, and the labels of the rules/equations that have been applied in $\mathcal{T}_{bank}$ and $\mathcal{T}^{\bullet}_{bank}$.

It is worth noting that the trace slice $\mathcal{T}^{\bullet}_{bank}$ greatly simplifies the trace $\mathcal{T}_{bank}$ by deleting all the bank accounts and account operations that are not related to the client `C` as well as all the internal flat/unflat rewrite steps, which are needed to implement rewriting modulo associativity and commutativity. In fact, the computed reduction rate is 83%, which clearly shows the drastic simplification that we have obtained.

Now, a quick inspection of $\mathcal{T}^{\bullet}_{bank}$ allows the existence of a misbehaving account operation to be recognized. Specifically, state `9` in the trace slice $\mathcal{T}^{\bullet}_{bank}$ has been obtained by reducing the term `debit(C, 50)` by means of the rule `debitERR` even though the current balance of `C` was only `20`. This suggests to us that `debit_ERR` might be faulty since it does not conform to its intended semantics, which forbids any withdrawal greater than the current funds available.

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

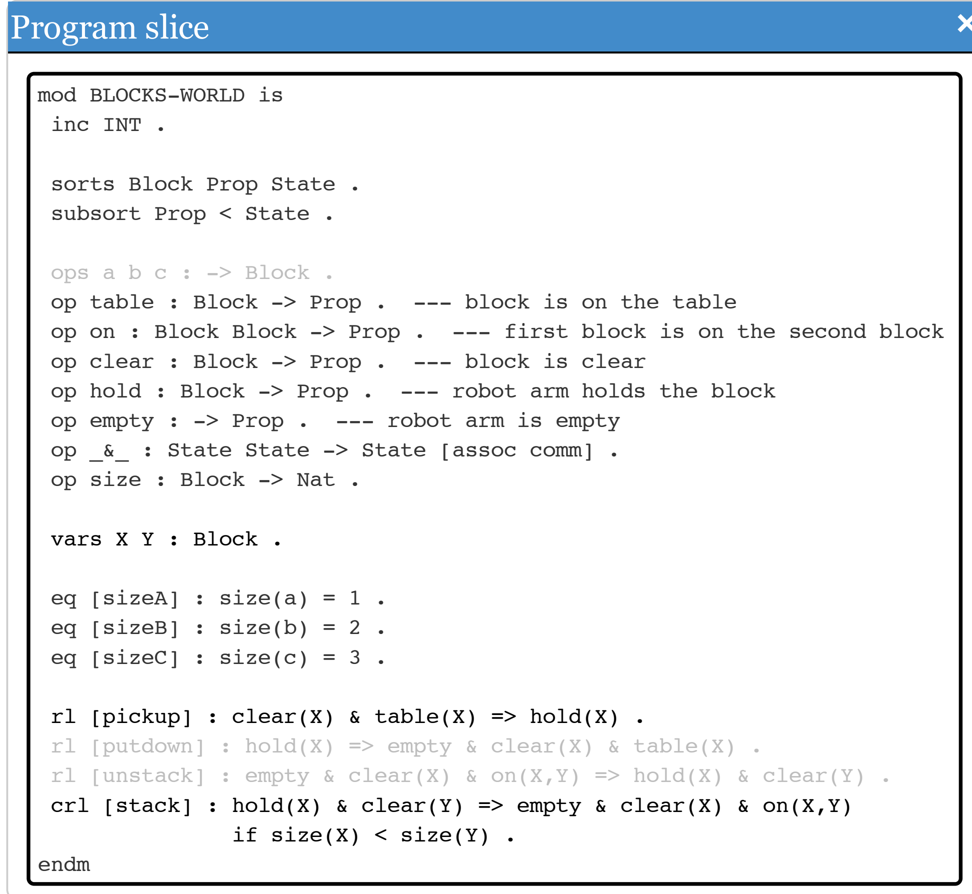
```
mod BLOCKS-WORLD is
  inc INT .

  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 2.4: `BLOCKS-WORLD` faulty Maude specification.

Example 2.1.2

Consider the Maude program `BLOCKS-WORLD` of Figure 2.4, which is a faulty mutation of one of the most popular planning problems in artificial intelligence. We assume that there are some blocks, placed on a table, that can be moved by means of a robot arm; the goal of the robot arm is to produce one or more vertical stacks of blocks. In our specification, 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 $\texttt{prop}_1$ `&` $\texttt{prop}_2$ `&` ... `&` $\texttt{prop}_\texttt{n}$, which describe any possible configuration of the blocks on the table as well as the status of the robot arm.



Program slice

```
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: Program slice computed w.r.t. the slicing criterion `empty & empty`.

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 B_1 from being piled on a block B_2 if the size of B_1 is greater than the size of B_2.

Barely perceptible, the Maude specification of Figure 2.4 fails to pro-

vide 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

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

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

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

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

To find the cause of this wrong behavior, we feed *i*JULIENNE with the faulty rewrite sequence $\mathcal{T} = \mathtt{s_i} \to^* \mathtt{s_f}$, and we initially slice $\mathcal{T}$ w.r.t. the slicing criterion that observes the two anomalous occurrences of the `empty` property in State $\mathtt{s_f}$. This task can be easily performed in *i*JULIENNE by first highlighting the terms that we want to observe in State $\mathtt{s_f}$ with the mouse pointer and then starting the slicing process. *i*JULIENNE yields a trace slice that simplifies the original trace by recording only those data that are strictly needed to produce the considered slicing criterion. Also, it automatically computes the corresponding program slice, which consists of the equations defining the `size` operator together with the `pickup` and `stack` rules (see Figure 2.5). 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 the search for a possible explanation for the wrong behavior. During this phase, we focus our attention on State 5 (Figure 2.6), which is inconsistent since it models a robot arm that is holding a block and is empty at the same time. Therefore, we decide to further refine the trace slice by incrementally applying backward trace slicing to State 5 w.r.t. the slicing criterion `hold( • ) & empty`. This way, we achieve a supplementary reduction of the previous trace slice in which we can easily observe in states 2 and 3 that the `hold` property only depends on the `clear` and `table` properties (see Figure 2.7).

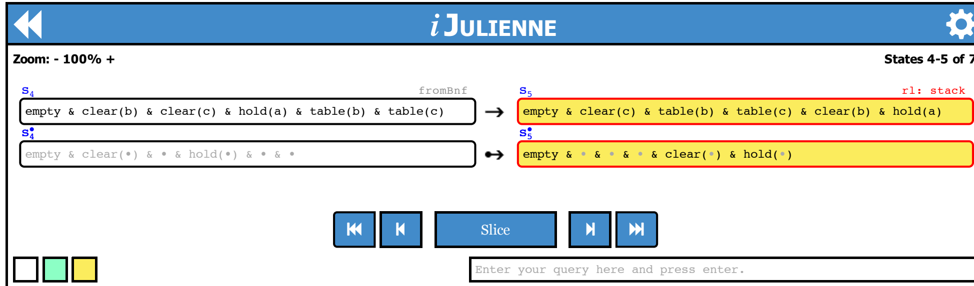


Figure 2.6: Navigation through the trace slice of the *Blocks World* example.

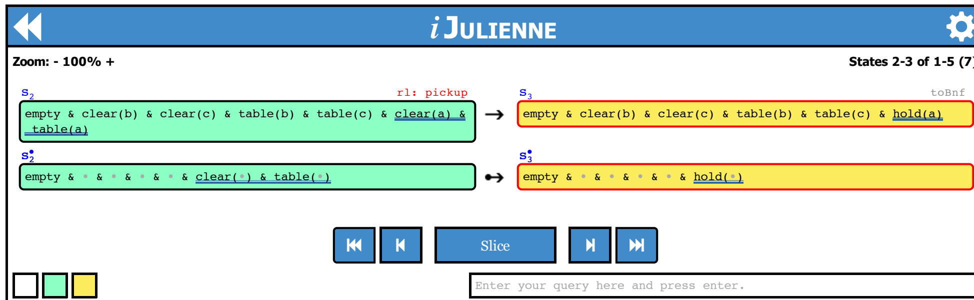


Figure 2.7: Navigation though the refined trace slice of the *Blocks World* example.

Furthermore, the computed program slice includes the single `pickup` rule (see Figure 2.8). 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 the `hold` property only relies on the `clear` and `table` 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

Program slice

```
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 2.8: Program slice computed w.r.t. the slicing criterion `hold(•)`.

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

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

2.1.2 Trace Querying with *i*JULIENNE

Execution traces of programs are a helpful source of information for program comprehension. However, they provide such a low-level picture of

program execution that users may experience several difficulties in interpreting and analyzing them. Trace querying [Duc99] allows a given execution trace to be analyzed at a higher level of abstraction by selecting only the subtrace that the user considers relevant.

Trace querying of Maude execution traces is naturally supported and completely automated by *i*JULIENNE. Indeed, execution traces can be simply queried by providing a slicing criterion (in the form of a filtering pattern) that specifies the target symbols the user decides to monitor. Hence, backward trace slicing is performed w.r.t. the considered criterion to compute an abstract view (i.e., the trace slice) of the original execution trace that only includes the information that is strictly required to yield the target symbols under observation. This way, users can focus their attention on the monitored data, which might otherwise be overlooked in the concrete trace.

Moreover, backward trace slicing correctness provides, for free, a means to understand the program behavior w.r.t. partially-defined inputs since the computed compatibility condition constrains the possible values that non-relevant inputs (modeled by $\bullet$-variables) might assume. In other words, a trace slice $\mathcal{T}^{\bullet}$ can be thought of as an intensional representation of possible concrete traces, which are compatible concretizations of $\mathcal{T}^{\bullet}$, that lead to the production of the monitored target symbols.

Example 2.1.3

The Maude specification of Figure 2.9, inspired by a similar one in [LS03], specifies the operator `minmax` that takes as input a list of natural numbers `L` and computes a pair `(m,M)` where `m` (resp., `M`) is the minimum (resp., maximum) of `L`.

Let $\mathcal{T}_{minmax}$ be the execution trace that reduces the input term `minmax(4 ; 7 ; 0)` to the normal form `PAIR(0,7)`. Now, assume that we are only interested in analyzing the subtrace that generates `0` in `PAIR(0,7)` —i.e., the minimum of the list `4 ; 7 ; 0`.

Thus, we can query $\mathcal{T}_{minmax}$ by specifying the pattern `PAIR(?,_)` that allows us to trace back only the first argument of `PAIR`, while the second one is discarded. *i*JULIENNE generates the trace slice $\mathcal{T}^{\bullet}_{minmax}$, given in Figure 2.10, whose compatibility condition is $\mathtt{B}^{\bullet} = \bullet_1 > \mathtt{0} \wedge \bullet_3 > \mathtt{0}$. The slice $\mathcal{T}^{\bullet}_{minmax}$ isolates all and only those function calls in the trace $\mathcal{T}$ that must be reduced to yield the minimum of the list `4 ; 7 ; 0`. Now,

```
mod MINMAX is
  inc INT .

  sorts List Pair .
  subsorts Nat < List .

  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 _;_ : List List -> List [ctor assoc] .

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

  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)))) .
  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 .
endm
```

Figure 2.9: Maude specification of the `minmax` function.

by analyzing the trace slice, it is immediate to see that the operators `2nd` and `Max` do not affect the observed result since they are not used in the trace slice. Also, by the correctness of our slicing technique, we can state that there are concrete instances $\mathtt{L_c}$ of the partially-defined input $\bullet_1$ `;` $\bullet_3$ `; 0` that meet the compatibility condition $\mathtt{B}^\bullet$ such that the minimum computed by the call `minmax`($\mathtt{L_c}$) will be `0`.

The `MINMAX` program is paradigmatic of a strategy known as introduction of mutual recursion, which separates a single function yielding pairs into the pair of its component functions (slices) [VO01]. Thus in a sense, the same effect (trace slicing) could be obtained for this example by slicing the source code in Maude before trace inspection.

In the following example, the trace querying capabilities of *i*JULIENNE are used to simplify the counter-examples traces provided by Web-TLR [ABER10], a RWL-based tool that allows real-size web applications to be formally specified and verified by using the built-in Maude model-checker, thus facilitating their analysis. In Web-TLR, a web applica-

Trace information

State	Label	Trace	Trace Slice
1	'Start	minmax(4 ; 7 ; 0)	minmax(•1 ; •3 ; 0)
2	fromBnf	minmax(4 ; 7 ; 0)	**minmax(•1 ; •3 ; 0)**
3	minmax2	PAIR(Min(4,1st(minmax(7 ; 0))),Max(4,2nd(minmax(7 ; 0))))	**PAIR(Min(•1,1st(minmax(•3 ; 0))),•0)**
4	minmax2	PAIR(Min(4,1st(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))),Max(4,2nd(minmax(7 ; 0))))	**PAIR(Min(•1,1st(PAIR(Min(•3,1st(minmax(0))),•10))),•0)**
5	1st	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(minmax(7 ; 0))))	PAIR(Min(•1,Min(•3,1st(minmax(0)))),•0)
6	minmax2	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,2nd(PAIR(Min(7,1st(minmax(0))),Max(7,2nd(minmax(0)))))))	PAIR(Min(•1,Min(•3,1st(minmax(0)))),•0)
7	2nd	PAIR(Min(4,Min(7,1st(minmax(0)))),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(•1,Min(•3,1st(minmax(0)))),•0)**
8	minmax1	PAIR(Min(4,Min(7,1st(PAIR(0,0)))),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(•1,Min(•3,1st(PAIR(0,•6)))),•0)**
9	1st	PAIR(Min(4,Min(7,0)),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(•1,Min(•3,0)),•0)**
10	Min1	PAIR(Min(4,0),Max(4,Max(7,2nd(minmax(0)))))	**PAIR(Min(•1,0),•0)**
11	Min1	PAIR(0,Max(4,Max(7,2nd(minmax(0)))))	PAIR(0,•0)
12	minmax1	PAIR(0,Max(4,Max(7,2nd(PAIR(0,0)))))	PAIR(0,•0)
13	2nd	PAIR(0,Max(4,Max(7,0)))	PAIR(0,•0)
14	Max1	PAIR(0,Max(4,7))	PAIR(0,•0)
15	Max2	PAIR(0,7)	**PAIR(0,•0)**
	Compatibility condition:	•1 > 0 and •3 > 0	
Total size:		679	219
Reduction Rate: 67%			

Figure 2.10: *i*JULIENNE output for the trace $\mathcal{T}^{\bullet}_{minmax}$ w.r.t. the slicing criterion automatically inferred in the query `PAIR(?,_)`.

tion is formalized by means of a Maude specification and then checked against a property specified in the *Linear Temporal Logic of Rewriting* (LTLR [Mes08]), which is a temporal logic specifically designed to model-check rewrite theories. A property is refuted by the LTLR model-checker by issuing a counter-example in the form of a rewrite sequence that reveals some undesired, erroneous behavior. Our example reproduces a typical trace analysis session that operates on the Maude specification of a complex webmail application checked with Web-TLR.

Example 2.1.4

Consider the webmail application written in Maude given in [ABER10] that models both server-side aspects (e.g., web script evaluations, database interactions) and browser-side features (e.g., forward/backward navigation, web page refreshing, window/tab openings). The web application behavior is formalized by using rewrite rules of the form [label] :

PROVIDE THE MAUDE INPUT PROGRAM AND EXECUTION TRACE ?

Webmail application | Upload

```
vars ba : BrowserActions .
vars ln tab f5 : Qid .

op T : Int Qid -> Tab [ctor] .
op tab-empty : -> Tab .
op _;_ : Tab Tab -> Tab [ctor assoc comm id: tab-empty] .

op f5-empty : -> Qid .
op _;_ : Qid Qid -> Qid [ctor assoc comm id: f5-empty] .

op bra-empty : -> BrowserActions .
```

Maude term of sort Trace | Generate

```
{'`[_`]_`[_`]`[_`][_]['B['idA.Id,'idw1.Id,''noPage.Qid,'_?_[''Welcome.Qid,'query-empty
.Query],
    'session-empty.Session,'_:_['_/_[''idEmail.Qid,'"email2".String],'_/_[''pass.Qi
d,'"secretAlice".String],
    '_/_[''user.Qid,'"alice".String]],'mes-empty.Message,'history-empty.History,'s_
```

Figure 2.11: Loading the webmail execution trace.

webState $\Rightarrow$ webState, where webState is a triple that we represent with the following operator $_\|_\|_$: (Browsers $\times$ Message $\times$ Server) $\rightarrow$ 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 web state transitions that represents a possible execution of the webmail application.

The trace slicing session starts by loading the Maude webmail specification, together with the execution trace to be analyzed, into the *i*JULIENNE trace analyzer. The trace can be directly pasted in the input form or uploaded from a trace file that was written off-line. It can also be

Query information ×

States where the query B(idA,_,?,_,_,_,_,_) was satisfied:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97			

The relevant data have been automatically inferred according to the provided query.
You can add/change the inferred data in the selected state:

```
[ B(idA,idw1,'Welcome, 'Home ?  'pass '= ""  :  'user '= ""  ,session-empty, 'idEmail / "em
ail2"  :  'pass / "secretAlice"  :  'user / "alice" ,m(idA,idw1, 'Welcome ? query-empty ,1)
,history-empty,1) ] bra-empty [ mes-empty ][ S(( 'Admin-Logout , updateDB(s( "adminPage" ),
s( "free" )) ,{ ( TRUE => 'Home ) },{ nav-empty }) : ( 'Administration ,  'adminPage := sel
ectDB(s( "adminPage" ))  ; if  'adminPage = s( "free" )  then updateDB(s( "adminPage" ),get
Session(s( "user" ))) ; setSession(s( "adminPage" ),s( "free" )) else setSession(s( "adminP
age" ),s( "busy" )) fi ,{ (  s( "adminPage" ) '== s( "busy" )  => 'Home ) },{ ( TRUE ->  'A
dmin-Logout ? query-empty  ) }) : ( 'Change-account , skip ,{ cont-empty },{ ( TRUE ->  'Ho
me ?  'newPass '= ""  :  'newUser '= ""   ) }) : ( 'Email-list ,  'u := getSession(s( "user
" ))  ;  'es := selectDB( 'u '. s( "-email" ) )  ; setSession(s( "idEmails-found" ),'es) ,{
 cont-empty },{ ( TRUE ->  'Home ? query-empty  ) : ( TRUE ->  'View-email ?  'idEmail '= "
"   ) }) : ( 'Home ,  'login := getSession(s( "login" ))  ; if  'login = null  then  'u :=
getQuery('user)  ;  'p := getQuery('pass)  ;  'p1 := selectDB('u)  ; if  'p = 'p1  then  'r
 := selectDB( 'u '. s( "-role" ) )  ; setSession(s( "user" ),'u) ; setSession(s( "role" ),'
```

Figure 2.12: Querying the webmail trace w.r.t. `B(idA,_,?,_,_,_,_,_)`.

dynamically computed by the system (using Maude meta-search capabilities) by introducing the initial and final states of the trace. In Figure 2.11, 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

Figure 2.13: Webmail trace slice after querying the trace.

`B(idA,_,?,_,_,_,_,_,_)`, which allows us to select all states in the trace that contain a browser data structure that is associated with user `idA`. Note that only the third argument of the browser data structure, which corresponds to the web page displayed on the browser, is declared relevant in the query (i.e., it is marked by the card `?`). Indeed, we are interested in tracking only the web pages visited by the user `idA`. The remaining information (such as script evaluation, and web interactions with other users) is not pinpointed and, hence, will be systematically removed by the slicing tool which facilitates comprehension. By running the query, *i*JULIENNE computes the outcome shown in Figure 2.12, 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 *Slice* button, we get a browsable trace slice (see Figure 2.13) 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.

2.1.3 Dynamic Program Slicing

As we illustrated in Figures 2.5 and 2.8, for the `BLOCKS-WORLD` example, by running *i*JULIENNE one is able to obtain not only a more compact and focused trace that corresponds to the execution of the program, but also to uncover statement dependencies that connect computationally related parts of the program. Hence a single walk over such dependencies is sufficient to implement a form of dynamic program slicing.

Program slicing [Wei81] is the computation of the set of program statements, the program slice, that may affect the values at some point of interest. A program slice consists of a subset of the statements of the original program, sometimes with the additional constraint that a slice must be a syntactically valid, executable program. Relevant applications of slicing include software maintenance, optimization, program analysis, and information flow control. An important distinction holds between static and dynamic slicing: whereas static slicing is performed with no other information than the source code itself, dynamic program slicing works on a particular execution of the program (i.e., a given execution trace) [KL88], hence it only reflects the actual dependencies of that execution, resulting in smaller program slices than static ones. Dynamic slicing is usually achieved by dynamic data-flow analysis along the program execution path. Although dynamic program slicing was first introduced to aid in user level debugging to locate sources of errors more easily, applications aimed at improving software quality, reliability, se-

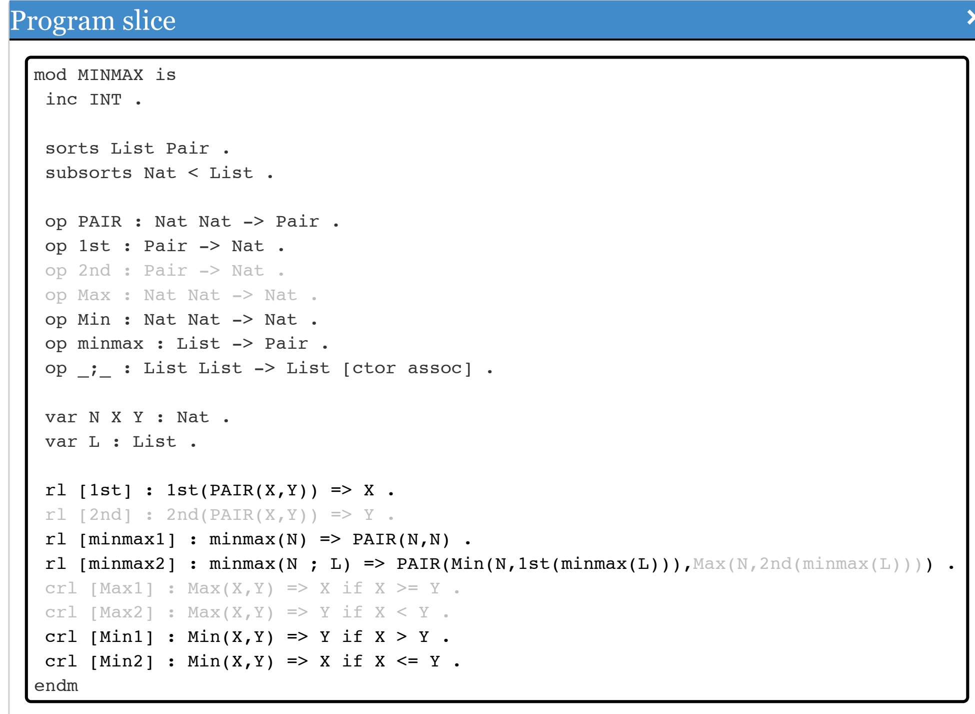


Figure 2.14: `MINMAX` program slice computed w.r.t. the query `PAIR(?,_)`.

curity and performance have also been identified as candidates for using dynamic program slicing.

Let us show how backward trace slicing can support the generation of dynamic program slices by detecting unused program fragments in a given trace slice.

Example 2.1.5

Consider the trace slice $\mathcal{T}^{\bullet}_{minmax}$ of Example 2.1.3 for the execution trace

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

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

- the unneeded operator declarations in the program signature;

- the rule definitions of the functions `2nd` and `Max`;
- the function calls to `2nd` and `Max` in the right-hand side of the `minmax2` rule.

Figure 2.14 shows the resulting dynamic program slice.

2.2 Experimental Evaluation

*i*Julienne is the first slicing-based, incremental trace analysis tool for Rewriting Logic that greatly reduces the size of the computation traces and can make their analysis feasible even for complex, real-size applications. *i*Julienne conveys an incremental slicing approach where the user can refine the slicing criteria and then the extra redundancies (i.e., the difference of the slices) are automatically done away with. It is important to note that our trace analyzer does not remove any information that is relevant, independently of the skills of the user. We have experimentally evaluated our tool in several case studies that are available at the *i*Julienne web site [iJu12] and within the distribution package, which also contains a user guide, the source files of the slicer, and related literature.

To properly assess the performance and scalability, we have tested *i*Julienne on several execution traces of increasing complexity. More precisely, we have considered

- two execution traces that model two runs of a fault-tolerant client-server communication protocol (`FTCP`) specified in Maude. Trace slicing has been performed according to two chosen criteria that aim at extracting information related to a specific server and client in a scenario that involves multiple servers and clients, and tracking the response generated by the server according to a given client request.
- two execution traces that model two instances of a well-known man-in-the-middle attack to the Needham-Schroeder network authentication protocol [Low96]. Specifically, they consist of a sequence of rewrite steps that represents the messages exchanged among three entities: an initiator A, a receiver B, and an intruder I that imitates

Example trace	Original trace size	Slicing criterion	Sliced trace size	% reduction	% reduction by changing criterion
FTCP.$\mathcal{T}_1$	2054	O_1(51.42%)	294	85.69%	97.89%
		O_2(40.01%)	316	84.62%	97.30%
FTCP.$\mathcal{T}_2$	1286	O_1(42.59%)	135	89.40%	98.11%
		O_2(20.37%)	97	92.46%	99.01%
NS-P.$\mathcal{T}_1$	21265	O_1(43.19%)	2249	89.42%	98.12%
		O_2(41.92%)	2261	89.36%	98.03%
NS-P.$\mathcal{T}_2$	34681	O_1(50.02%)	3015	91.30%	99.08%
		O_2(47.64%)	3192	90.79%	98.84%
web-TLR.$\mathcal{T}_1$	38983	O_1(2.93%)	2045	94.75%	99.28%
		O_2(8.09%)	2778	92.87%	99.14%
web-TLR.$\mathcal{T}_2$	69491	O_1(18.57%)	8493	87.78%	97.99%
		O_2(14.11%)	5034	92.76%	99.05%
SmallKb.$\mathcal{T}_1$	4149	O_1(44.25%)	459	88.94%	98.08%
		O_2(39.28%)	389	90.62%	98.26%
SmallKb.$\mathcal{T}_2$	5718	O_1(29.83%)	419	92.67%	98.89%
		O_2(31.53%)	618	89.19%	98.15%
			% reduction average	90.16%	98.45%

Table 2.1: Incremental slicing benchmarks.

A to establish a network session with B. The chosen slicing criteria selects the intruder's actions as well as the intruder's knowledge at each rewrite step discarding all the remaining session information.

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

- two execution traces generated by the Pathway Logic Maude implementation. Pathway Logic [Tal08] is a RWL-based approach to modeling biological entities and processes that formalizes the ordinary models that biologists commonly use to explain biological processes. Roughly speaking, Pathway logic models are Maude executable specifications whose execution traces provide a rewriting-based description of metabolic pathways. The traces that we consider in our experiments model responses to signal stimulation in epithelial-like cells. The chosen criteria allow one to detect cause and effect relations (e.g., the signal responsible for the production

of an observed chemical) and select only those chemical reactions that are involved in actions of interest (e.g., protein complexing, phosphorylation).

The results of our experiments are shown in Table 2.1 and the source code of the benchmark examples is given in Appendix A. The execution traces for the considered cases consist of sequences of 10–1000 states, each of which contains from 60 to 5000 characters. In column **Slicing criterion** we indicate the size of each criterion as a percentage of the last state of the trace, which ranges between ∼3% and ∼51%. In all the experiments, not only the trace slices that we obtained show impressive reduction rates (ranging from ∼85% to ∼95%), but we were also even able to strikingly improve these rates by an average of 8.5% (ranging from ∼97% to ∼99%) by using *incremental slicing*. In most cases, the delivered trace slices were cleaned enough to be easily analyzed, and we noted an increase in the effectiveness of the analysis processes. Other benchmark programs we have considered are available at the *i*Julienne web site [iJu12].

With regard to the time required to perform the analyses, our implementation is rather time efficient; the elapsed times are small even for very complex traces and scale linearly. For example, running the slicer for a 20Kb trace in a Maude specification with about 150 rules and equations –with AC rewrites– took less than 1 second (480.000 rewrites per second on standard hardware, 2.26GHz Intel Core 2 Duo with 4Gb of RAM memory).

Part II

Forward Trace Analysis

Chapter 3

Exploring Conditional Rewriting Logic Computations

In this chapter, we present a rich and highly dynamic, parameterized technique for the inspection of RWL computations that allows the nondeterministic execution of a given *conditional* rewrite theory to be followed up in different ways. With this technique, an analyst can browse, slice, filter, or search the execution traces as they come to life during the program execution. The navigation of the execution trace is driven by a user-defined, inspection function that specifies the required exploration mode. By selecting different inspection functions, one can automatically derive a family of practical algorithms such as program steppers and more sophisticated dynamic trace slicers that compute summaries of the computation tree, thereby facilitating the dynamic detection of control and data dependencies across the tree. Our methodology allows users to evaluate the effects of a given statement or instruction in isolation, track input change impact, and gain insight into program behavior (or misbehavior).

Our generic scheme basically consists of:

1) a generic inspection technique that allows instrumented traces to be inspected according to a given modality, and

2) a generic, slicing-based exploration technique for (instrumented) computation trees that allows the user to incrementally generate and inspect a fragment of the computation tree.

3.1 The Generic Exploration Scheme

Given a conditional rewrite theory $\mathcal{R}$, the transition space of all computations in $\mathcal{R}$ from the initial term s can be represented as a *computation tree*,[1] $\mathcal{T}_{\mathcal{R}}(s)$. RWL computation trees are typically large and complex objects to deal with because of the highly-concurrent, nondeterministic nature of rewrite theories. Also, their complete generation and inspection are generally not feasible since some of their branches may be infinite as they encode nonterminating execution traces.

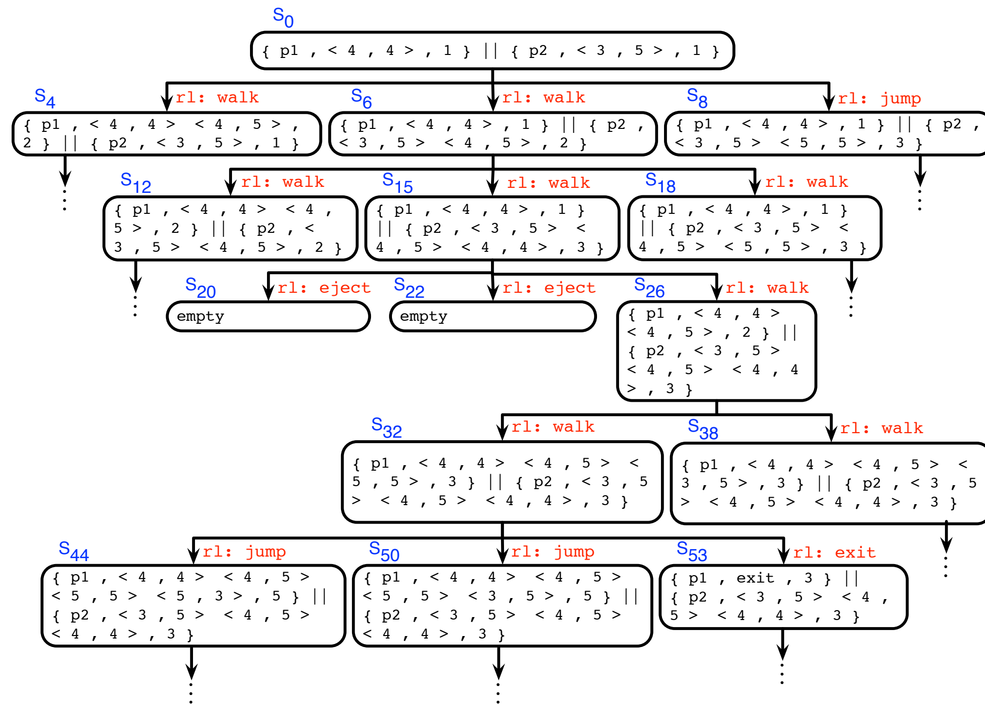


Figure 3.1: Computation tree.

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

Example 3.1.1

Consider the rewrite theory of Example 0.3.1 together with the system state

`{p1,< 4,4 >,1} || {p2,< 3,5 >,1}`

that specifies two initial configurations for two players `p1` and `p2` in the maze. In this case, the computation tree describes all of the possible trajectories that the two players `p1` and `p2` (respectively starting at positions `< 4,4 >` and `< 3,5 >`) can take when they move simultaneously in the same maze. The paths are built by repeated (and independent) applications of the `walk` and `jump` rules, while the `eject` and `exit` rules respectively implement the expulsion of colliding players and the output of game players who reach the exit. A fragment of the computation tree is shown in Figure 3.1. For simplicity, we have chosen to decorate tree edges only with the labels of the rules that have been applied at each rewrite step, while other information such as the computed substitution and the rewrite position are omitted in the depicted tree.

The instrumented version of a computation tree $\mathcal{T}_{\mathcal{R}}(s)$ can be constructed from $\mathcal{T}_{\mathcal{R}}(s)$ by expanding each execution trace in $\mathcal{T}_{\mathcal{R}}(s)$ into its corresponding instrumented counterpart as explained in Section 0.5. Also, it is possible to switch from the instrumented computation tree to the non-instrumented one by simply hiding the intermediate B-matching transformations and built-in evaluations that occur in the instrumented tree. In the sequel, we let $\mathcal{T}^{+}_{\mathcal{R}}(s)$ denote the instrumented computation tree that originates from the state s.

In the following section we formalize the generic inspection technique.

3.1.1 Inspecting the Instrumented Traces

Let us first introduce the notion of inspection function.

Definition 3.1.2 (Inspection Function) *An* inspection function *is a function* $\mathcal{I}(s^\bullet, s \to_K t)$ *that, given a rewrite step* $s \to_K t$ *and a term slice* $s^\bullet$ *of* s*, computes a term slice* $t^\bullet$ *of* t*.*

Roughly speaking, inspection functions allow us to control the information content conveyed by term slices resulting from the execution of

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

Figure 3.2: The inference rule trans of the transition system $(\mathit{Conf}, \Longrightarrow)$.

$\to_K$-rewrite steps. It is worth noting that distinct implementations of the inspection functions may produce distinct slices of the considered rewrite step. Several examples of inspection functions are discussed in Chapter 4. We assume that the special value fail is returned by the inspection function whenever no slice $t^\bullet$ can be computed by $\mathcal{I}$.

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

Let us formalize a calculus that allows the generation of instrumented trace slices w.r.t. $\mathcal{I}$ by means of a transition system $(\mathit{Conf}, \Longrightarrow)$ where:

- *Conf* is a set of *configurations* of the form $\langle \mathcal{T}, \mathcal{F}^\bullet\rangle$, where $\mathcal{T}$ is an instrumented execution trace and $\mathcal{F}^\bullet$ is an instrumented trace slice;

- the transition relation $\Longrightarrow$ implements the calculus of instrumented trace slice generation and is the smallest relation that satisfies the inference rule trans given in Figure 3.2.

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

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

Definition 3.1.3 (Instrumented Trace Slice w.r.t. $\mathcal{I}$) *Given the instrumented execution trace $\mathcal{T} = (s_0 \to_K s_1 \to_K \cdots \to_K s_n)$, with $n \geq 1$, the* instrumented trace slice *$\mathcal{T}^\bullet$ of $\mathcal{T}$ w.r.t. the inspection function $\mathcal{I}$ and term slice $s_0^\bullet$ of s_0 is defined by the function $Cslice(s_0^\bullet, \mathcal{T}, \mathcal{I})$, which is defined as follows.*

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

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

In the following section we formulate our generic, slicing-based exploration technique that allows the user to incrementally generate and inspect a fragment of the instrumented computation tree $\mathcal{T}_\mathcal{R}^+(s)$ by expanding (slices of) its computation states into their descendants starting from the root node. The exploration is an interactive procedure that can be completely controlled by the user, who is free to choose the computation states to be expanded.

3.1.2 Exploring the Instrumented Computation Tree Slices

Instrumented computation tree slices are formally defined as follows.

Definition 3.1.4 (Instrumented Computation Tree Slice) *Let $\mathcal{T}_\mathcal{R}^+(s_0)$ be an instrumented computation tree for the term s_0 in the conditional rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$; let $s_0^\bullet$ be a term slice of s_0; and let $\mathcal{I}$ be an inspection function. An instrumented computation tree slice for $s_0^\bullet$ in $\mathcal{R}$ w.r.t. $\mathcal{I}$ is a tree $\mathcal{T}_{\mathcal{R},\mathcal{I}}^+(s_0^\bullet)$ (simply denoted by $\mathcal{T}_\mathcal{R}^+(s_0^\bullet)$ when no confusion can arise) such that:*

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

```
function expand(s, s•, R, I)
1. A = ∅
2. for each M ∈ mS(s)
3.   M• = Cslice(s•, instrument(M), I)
4.   if M• ≠ nil then A = A ∪ {M•}
5. end
6. return A
endf
```

Figure 3.3: The one-step *expand* function.

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

In our methodology, instrumented computation tree slices are incrementally constructed by expanding tree nodes (i.e., term slices), starting from the root node —i.e, the initial term slice $s^\bullet$ which can be generated by invoking the *Tslice* function on the input term s and a set of user-defined, relevant positions P of s (in symbols, $s^\bullet = \mathit{Tslice}(s, P)$). Formally, given the term s and the term slice $s^\bullet$ of s, the expansion of s in the rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ w.r.t. the inspection function $\mathcal{I}$ is defined by the function $\mathit{expand}(s, s^\bullet, \mathcal{R}, \mathcal{I})$ of Figure 3.3, which unfolds the term slice $s^\bullet$ by deploying and then slicing all the possible instrumented Maude computation steps stemming from s that are given by $m\mathcal{S}(s)$. In other words, for each Maude step $\mathcal{M} = s \rightarrow^*_{\Delta,B} s{\downarrow_{\Delta,B}} \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$, we first compute its instrumented version and then the corresponding instrumented Maude step slice $\mathcal{M}^\bullet$ is generated, which is finally added to the set $\mathcal{A}$.

The overall construction methodology for instrumented computation tree slices is specified by the function *explore*, defined in Figure 3.4. Given a rewrite theory $\mathcal{R}$, a term slice $s_0^\bullet$ of the initial term s_0, and an inspection function $\mathcal{I}$, the function *explore* essentially formalizes an interactive procedure that is driven by the user starting from an elemental tree slice fragment, which only consists of the sliced root node $s_0^\bullet$. The instrumented computation tree slice $\mathcal{T}^+_{\mathcal{R}}(s_0^\bullet)$ is built by choosing, at each loop iteration of the algorithm, the tree leaf that repre-

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

endf

Figure 3.4: The interactive *explore* function.

sents the term slice to be expanded by means of the auxiliary function $\mathit{pickLeaf}(\mathcal{T}_{\mathcal{R}}^{+}(s_0^\bullet))$, which allows the user to freely select a leaf node from the frontier of the current tree $\mathcal{T}_{\mathcal{R}}^{+}(s_0^\bullet)$. Then, $\mathcal{T}_{\mathcal{R}}^{+}(s_0^\bullet)$ is augmented by calling $\mathit{addPaths}(\mathcal{T}_{\mathcal{R}}^{+}(s_0^\bullet), s^\bullet, \mathit{expand}(\mathit{unhide}(s^\bullet), s^\bullet, \mathcal{R}, \mathcal{I}))$, where $\mathit{unhide}(s^\bullet)$ recovers the original term s from $s^\bullet$. This function call adds all the instrumented trace slices w.r.t. $\mathcal{I}$ and $s^\bullet$ that correspond to the Maude steps that originate from the term s.

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

Chapter 4

Exploration Modalities

The methodology given in Section 3.1 provides a generic scheme for the exploration of (instrumented) computation trees w.r.t. a given inspection function $\mathcal{I}$ that must be selected or provided by the user. In this chapter, we show four implementations of the inspection function $\mathcal{I}$ that produce three distinct exploration strategies. In the first case, the considered inspection modality allows an interactive program stepper to be derived in which conditional rewriting logic theories can be step-wisely animated. In the second case, we implement a partial stepper that allows execution traces with partial input to be stepped. The third inspection modality formalizes an automated, forward trace slicing technique that simplifies the execution traces and allows relevant control and data information to be easily identified within the computation trees. Finally, by reusing the third inspection function, we show how we can also formalize backward trace slicing as an instance of the scheme. This is very interesting because it enables reusing large parts of the code that implements our exploration tool.

4.1 Interactive Stepper

Program animators have existed since the early years of programming. Although several steppers have been implemented in the functional programming community (see [CFF01] for references), none of these systems applies to the animation and dynamic forward slicing of Maude computations.

Given an instrumented computation tree $\mathcal{T}^{+}_{\mathcal{R}}(s_0)$ for an initial state s_0 and a conditional rewrite theory $\mathcal{R}$, the stepwise inspection of the computation tree can be directly implemented by instantiating the exploration scheme of Section 3.1 with the basic inspection function $\mathcal{I}_{step}(s, s \stackrel{r,\sigma,w}{\rightarrow}_K t) = t$, which simply returns the reduced term t of the rewrite step $s \stackrel{r,\sigma,w}{\rightarrow}_K t$. This way, by starting the exploration from a term slice that corresponds to the whole initial term s_0 (i.e., $s_0^{\bullet} = s_0$), the call $\mathit{explore}(s_0, s_0^{\bullet}, \mathcal{R},$

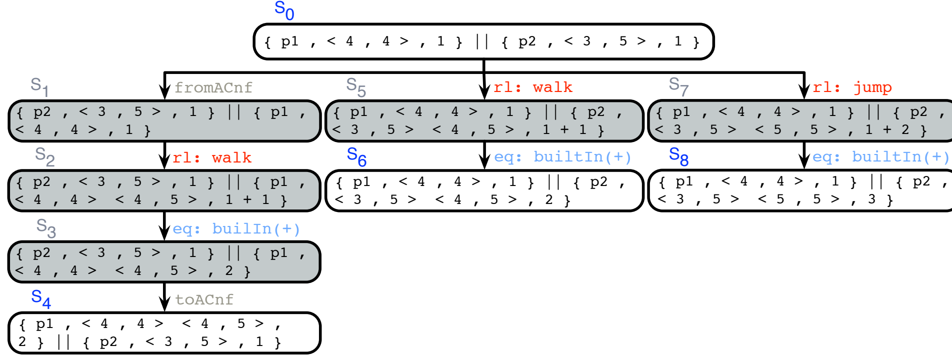


Figure 4.1: Inspection of the state s_0 w.r.t. $\mathcal{I}_{step}$.

$\mathcal{I}_{step}$) generates (a fragment of) the instrumented computation tree $\mathcal{T}^+_{\mathcal{R}}(s_0)$ whose topology depends on the program states that the user decides to expand during the exploration process.

Example 4.1.1

Consider the rewrite theory $\mathcal{R}$ in Example 0.3.1 and the computation tree in Example 3.1.1. Assume the user starts the exploration by calling $explore(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I}_{step})$, with $s_0 = s_0^\bullet$, which allows all the Maude steps that stem from the initial term s_0 to be expanded w.r.t. the inspection function $\mathcal{I}_{step}$. This generates the instrumented computation tree fragment $\mathcal{T}^+_{\mathcal{R}}(s_0)$ in Figure 4.1, where the instrumentation is made explicit. Now, the user can either quit or carry on with the exploration of nodes s_4, s_6, and s_8, which would result in the instrumented version of the tree fragment that is shown in Figure 3.1.

4.2 Partial Stepper

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

We define the following inspection function:

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

Roughly speaking, given a conditional rewrite step $s \stackrel{r,\sigma,w}{\to}_K t$, the inspection modality $\mathcal{I}_{pstep}$ returns a term slice $t^\bullet$ of the reduced term t, whenever $s^\bullet$ can be conditionally rewritten to $t^\bullet$ using the very same rule r at the same position w with the corresponding matching substitution $\sigma^\bullet$. The particularization of the exploration scheme given by the inspection modality $\mathcal{I}_{pstep}$ allows an interactive, partial stepper to be derived, in which the user can observe a distinguished part of the state, thereby producing more compact and focused representations of the visited fragment of the (instrumented) computation tree.

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

Example 4.2.1

Consider the computation tree of Example 3.1.1 and the initial state:

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

Let $s_0^\bullet = \{\texttt{p1,< 4,4 >,1}\} \texttt{ || } \bullet_1$ be a term slice of s_0 where only the triple of player `p1` is observed. Assume that the inspection function $\mathcal{I}_{pstep}$ is used to generate computation tree slice fragments. The computation tree slice fragment shown in Figure 4.2 is obtained by first expanding the node $s_0^\bullet$ into $s_3^\bullet$, then the node $s_3^\bullet$ into $s_6^\bullet$ and $s_9^\bullet$, then the node $s_9^\bullet$ into $s_{12}^\bullet$, and then the node $s_{12}^\bullet$ into $s_{15}^\bullet$ and $s_{17}^\bullet$. Note that the adopted partial stepping strategy allows a simplified view of (a part of) the considered computation tree to be constructed. More precisely, given the input encoded into the initial term slice $s_0^\bullet$, the computation can evolve by simply applying the rule `walk` to `p1`'s triple. By isolating `p1` movements in the tree slice fragment computed by partial stepping, the user can immediately observe and analyze why the player does not leave the game after reaching the exit but continues wandering for a while, which is an unnoticed side effect of declaring `exit` as a rule. To prevent `exit` from being nondeterministically applied in competition with other rules such as `walk` or `jump`, `exit` must be programmed as an equation to be

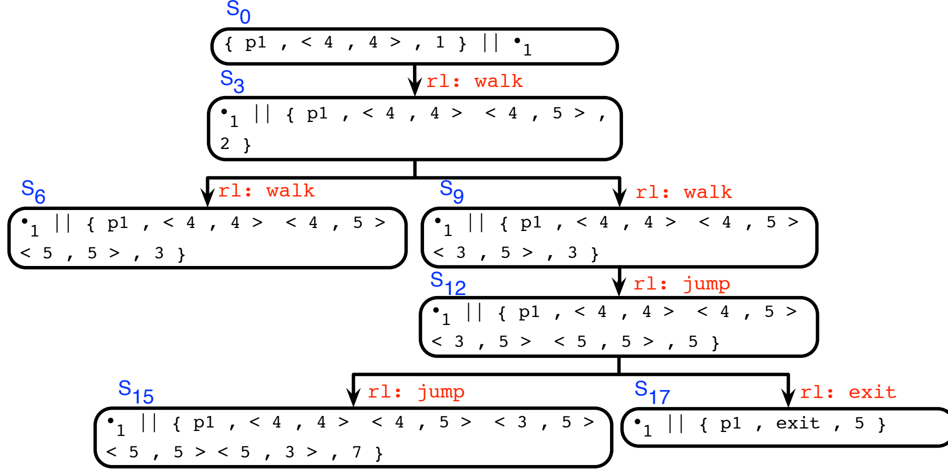


Figure 4.2: Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{pstep}$.

deterministically used for normalizing the system state after the player has reached the maze exit. Note also that the `eject` rule does not appear in the tree slice fragment since player `p2` has been filtered out from the initial term slice $s_0^\bullet$ and therefore `eject` cannot be applied. Indeed, two players are required to match the left-hand side of the `eject` rule.

It is worth noting that partial stepping can be very useful to approximate needed redexes [HL79] in orthogonal theories [HL91], they are those that make the computation flounder when replaced by $\bullet$ as illustrated in the following example.

Example 4.2.2

Consider the orthogonal theory composed of the two equations $[e_1]$: $f(X, 0) = X$ and $[e_2] : h(Z) = Z$. We can reduce to 0 the input expression $f(h(0), h(0))$ as follows:

$$f(h(0), h(0)) \xrightarrow{e_2} f(h(0), 0) \xrightarrow{e_1} h(0) \xrightarrow{e_2} 0$$

However, if we abstract any of the outermost redexes $h(0)$ by $\bullet$, then the corresponding partial computation flounders.

1. $f(h(0), \bullet) \stackrel{e_2}{\to} f(h(0), \bullet)$, and it flounders.
2. $f(\bullet, h(0)) \stackrel{e_2}{\to} f(\bullet, h(0)) \stackrel{e_1}{\to} \bullet$, and then flounders.

Hence, we safely conclude that both outermost redexes $h(0)$ are needed [DM97, Mid99].

4.3 Stepper and Partial Stepper Correctness

In this section, we provide a notion of correctness, which we call *universal correctness*, that holds for each instrumented trace slice computed by the stepper and the partial stepper described in Section 4.1 and Section 4.2, respectively. Universal correctness of an instrumented trace slice is formally defined as follows.

Definition 4.3.1 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \stackrel{r_1,\sigma_1,w_1}{\to}_K \cdots \stackrel{r_n,\sigma_n,w_n}{\to}_K s_n)$ be an instrumented execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}$ be an inspection function. Then, an instrumented trace slice $s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}$ and $s_0^\bullet$ is* universally correct *iff, for every instance s'_0 of $s_0^\bullet$, there exists an instrumented trace slice concretization $s'_0 \stackrel{r_1,\sigma'_1,w_1}{\to}_K \cdots \stackrel{r_n,\sigma'_n,w_n}{\to}_K s'_n$.*

Roughly speaking, Definition 4.3.1 ensures that, given an instrumented trace slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet)$, for every instance s'_0 of $s_0^\bullet$, there exists an instrumented trace slice concretization $\mathcal{T}'$ of $\mathcal{T}^\bullet$ in which the very same rules involved in $\mathcal{T}^\bullet$ can be applied again, at the same positions. This amounts to saying that we can reproduce all the relevant information of $\mathcal{T}^\bullet$ in any rewrite sequence that instantiates $\mathcal{T}^\bullet$.

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

Proposition 4.3.2 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \stackrel{r_1,\sigma_1,w_1}{\to}_K \cdots \stackrel{r_n,\sigma_n,w_n}{\to}_K s_n)$ be an instrumented execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}_{pstep}$ be the partial stepper inspection function. Then, every instrumented trace slice $s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$ is universally correct.*

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

Base case: $n = 1$. Let us consider the one-step, instrumented trace slice $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet$. By Definition 3.1.3, $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet$ has been obtained by calling the function $\mathit{Cslice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1, \mathcal{I}_{pstep})$. Hence, $\mathcal{T}_1^\bullet = (s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet) = \mathit{Cslice}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1, \mathcal{I}_{pstep})$. This implies that the following transition $\langle s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1, s_0^\bullet\rangle \Longrightarrow^* \langle \mathtt{nil}, s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet\rangle$ has been performed in the transition system $(\mathit{Conf}, \Longrightarrow)$ by means of one application of the trans rule. By definition of the trans rule (see Figure 3.2), $s_1^\bullet = \mathcal{I}_{pstep}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1)$. Now, by definition of $\mathcal{I}_{pstep}$, $s_1^\bullet = \mathcal{I}_{pstep}(s_0^\bullet, s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow_K} s_1)$ iff $s_0^\bullet \overset{r_1,\sigma^\bullet,w_1}{\rightarrow_K} s_1^\bullet$. To complete the proof, it suffices to observe that the rewriting relation $\rightarrow_K$ is stable (i.e., it is closed under substitution applications). This amounts to saying that $s_0^\bullet\sigma' \overset{r_1,\sigma^\bullet\sigma',w_1}{\rightarrow_K} s_1^\bullet\sigma'$, for every substitution σ'.

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

Inductive case: $n > 1$. Let us consider the instrumented trace slice $\mathcal{T}^\bullet = (s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet)$, with $n > 1$, w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$. By the induction hypothesis, the instrumented trace slice $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_{n-1}^\bullet$ w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$ is universally correct, that is, for every instance s_0' of $s_0^\bullet$, there exists a instrumented trace slice concretization:

$$s_0' \overset{r_1,\sigma_1',w_1}{\rightarrow_K} \cdots \overset{r_{n-1},\sigma_n',w_{n-1}}{\rightarrow_K} s_{n-1}' \tag{4.1}$$

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

$$s_{n-1}'' \overset{r_n,\sigma_n'',w_n}{\rightarrow_K} s_n'' \tag{4.2}$$

By choosing $s_{n-1}'' = s_{n-1}'$, we can glue together the rewrite sequences 4.1 and 4.2 thereby obtaining that, for every instance s_0' of

$s_0^\bullet$, there exists a instrumented trace slice concretization:

$$s'_0 \stackrel{r_1,\sigma'_1,w_1}{\rightarrow_K} \cdots \stackrel{r_{n-1},\sigma'_{n-1},w_{n-1}}{\rightarrow_K} s'_{n-1} \stackrel{r_n,\sigma'_n,w_n}{\rightarrow_K} s'_n \tag{4.3}$$

Hence, $\mathcal{T}^\bullet$ is a universally correct instrumented trace slice w.r.t. $\mathcal{I}_{pstep}$ and $s_0^\bullet$.

■

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

Corollary 4.3.3 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow_K} \cdots \stackrel{r_n,\sigma_n,w_n}{\rightarrow_K} s_n)$ be an instrumented execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$. Let $\mathcal{I}_{step}$ be the stepper inspection function. Then, $\mathcal{T}$ coincides with the instrumented trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{step}$ and $s_0^\bullet = s_0$.*

Proof. Immediate by the fact that s_0 is a term that does not contain $\bullet$-variables, so $\mathcal{I}_{step}$ behaves as $\mathcal{I}_{pstep}$ and generates a universally correct instrumented trace slice $s_0 \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n$ by Proposition 4.3.2. The slice is unique as $\mathcal{I}_{step}$ cannot introduce $\bullet$-variables in the instrumented trace slice that can be bound to arbitrary terms. Therefore, $\mathcal{T}^\bullet$ is the very same $\mathcal{T}$. ■

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

Theorem 4.3.4 (Universal Correctness) *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{I} \in \{\mathcal{I}_{step}, \mathcal{I}_{pstep}\}$. Let $s_0^\bullet$ be a term slice of term s_0. Let $\mathcal{T}^+_\mathcal{R}(s_0^\bullet)$ be (a fragment of) the instrumented computation tree slice in $\mathcal{R}$ w.r.t. $\mathcal{I}$ and $s_0^\bullet$ computed by the function explore$(s_0, s_0^\bullet, \mathcal{R}, \mathcal{I})$. Then, each branch $s_0^\bullet \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow s_n^\bullet$ in $\mathcal{T}^+_\mathcal{R}(s_0^\bullet)$ is a universally correct instrumented trace slice w.r.t. $\mathcal{I}$ and $s_0^\bullet$.*

Proof. Immediate. It suffices to apply Proposition 4.3.2 to each Maude step of each branch of $\mathcal{T}^{+}_{\mathcal{R}}(s^{\bullet}_0)$ when $\mathcal{I} = \mathcal{I}_{pstep}$ and to apply Corollary 4.3.3 when $\mathcal{I} = \mathcal{I}_{step}$. ∎

Note that universal correctness of an instrumented computation tree slice $\mathcal{T}^{+}_{\mathcal{R}}(s^{\bullet}_0)$ directly implies the universal correctness of the associated computation tree slice $\mathcal{T}_{\mathcal{R}}(s^{\bullet}_0)$. This is because $\mathcal{T}_{\mathcal{R}}(s^{\bullet}_0)$ is obtained from $\mathcal{T}^{+}_{\mathcal{R}}(s^{\bullet}_0)$ by simply removing those sliced steps that correspond to instrumentation transformations.

4.4 Forward Trace Slicer

Forward trace slicing is a program analysis technique that allows execution trace to be simplified w.r.t. a selected slice of their initial term. More precisely, given an instrumented execution trace $\mathcal{T}$ with initial term s_0 and a term slice $s^{\bullet}_0$ of s_0, forward slicing yields a simplified view $\mathcal{T}^{\bullet}$ of $\mathcal{T}$ in which each term s of the original instrumented execution trace is replaced by the corresponding term slice $s^{\bullet}$ that only records the information that depends on the meaningful symbols of $s^{\bullet}_0$, while irrelevant data are simply pruned away.

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

A precise formalization of the inspection function $\mathcal{I}_{slice}$ is provided by the function given in Figure 4.3. By adopting the inspection function $\mathcal{I}_{slice}$, the exploration scheme of Section 3.1 automatically turns into an interactive, forward trace slicer that expands computation states using the slicing methodology encoded into the inspection function $\mathcal{I}_{slice}$. In other words, given an instrumented computation tree $\mathcal{T}^{+}_{\mathcal{R}}(s_0)$ and a user-defined term slice $s^{\bullet}_0$ of the initial term s_0, any instrumented trace slice $s^{\bullet}_0 \bullet\!\!\to s^{\bullet}_1 \cdots \bullet\!\!\to s^{\bullet}_n$ in the tree $\mathcal{T}^{+}_{\mathcal{R}}(s^{\bullet}_0)$, which is computed by the *explore* function, is the sliced counterpart of an instrumented execution

```
function I_slice(s•, s →K^{r,σ,w} t)   /* Assuming: [r] : λ ⇒ ρ if C
 1. if w ∈ MPos(s•) then                    and C = c1 ∧ ... ∧ cn */
 2.    θ = {x/fresh• | x ∈ Dom(σ)}
 3.    ψ0 = (θmgu(λθ, (s•|w)))↾Dom(σ)
 4.    for i = 1 to n do
 5.      ψi = process-condition(ci, σ, ψi−1)
 6.    od
 7.    t• = s•[ρψn]w
 8. else
 9.    t• = s•[fresh•]w'    with w' ≤ w ∧ s•|w' = •i, for some i
10. fi
11. return t•
endf
```

Figure 4.3: Inspection function that models the forward slicing of a conditional rewrite step.

trace $s_0 \to_K s_1 \cdots \to_K s_n$ (w.r.t. the term slice $s_0^\bullet$) in the instrumented computation tree $\mathcal{T}^+_{\mathcal{R}}(s_0)$.

Roughly speaking, the inspection function $\mathcal{I}_{slice}$ works as follows. When the rewrite step $\mu : (s \stackrel{r,\sigma,w}{\to}_K t)$ occurs at a position w that is not a meaningful position of $s^\bullet$ (in symbols, $w \notin \mathcal{MP}os(s^\bullet)$), trivially μ does not contribute to producing the term slice $t^\bullet$. Actually, the rewriting position w might not even occur in $s^\bullet$, hence we consider the prefix w' of w that points to a $\bullet$-variable in $s^\bullet$, i.e., $s^\bullet_{|w'}$ is a $\bullet$-variable. This position exists and is unique. Now, since no new relevant information descends from the term slice $s^\bullet$, $\mathcal{I}_{slice}$ returns a variant $s^\bullet[\mathit{fresh}^\bullet]_{w'}$ of $s^\bullet$ where the subterm of $s^\bullet$ at the position w' has been replaced by a new fresh $\bullet$-variable that completely abstracts the contractum computed by μ.

Example 4.4.1

Consider the Maude specification of Example 0.3.1 and the following rewrite step μ : {`p1,< 4,4 >,1`} `||` {`p2,< 3,5 >,1`} $\stackrel{\texttt{walk}}{\to}_K$ {`p1,< 4,4 >,1`} `||` {`p2,< 3,5 > < 4,5 >,1+1`}. Let $s^\bullet$ = {`p1,< 4,4 >,1`} `||` $\bullet_1$ be a term slice of {`p1,< 4,4 >,1`} `||` {`p2,< 3,5 >,1`}. Since the rewrite step μ occurs at position $2 \notin \mathcal{MP}os($ {`p1,< 4,4 >,1`} `||` $\bullet_1$ $)$, which is not a

```
function process-condition(c, σ, ψ)
 1. case c of
 2.   (p := m) ∨ (m ⇒ p) :    /* matching conditions
 3.     if (mσ = pσ)              and rewrite expressions */
 4.       δ = mgu(p, mψ)
 5.     else
 6.       ((mσ)• •→+ (pσ)•) = Cslice(mψ, mσ →+_K pσ, I_slice)
 7.       δ = mgu(p, (pσ)•)
 8.     fi
 9.     return (δ ⇑ ψ)↾Dom(ψ)
 10.  e :   /* equational conditions */
 11.     return ψ
 12. end case
endf
```

Figure 4.4: The condition processing function.

meaningful position of $s^\bullet$, the inspection function $\mathcal{I}_{slice}$ returns the variant of {p1,< 4,4 >,1} || $\bullet_2$ of $s^\bullet$, where $\bullet_2$ is a fresh variable generated by the function *fresh*$^\bullet$.

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

More specifically, given the rewrite step $\mu : s \stackrel{r,\sigma,w}{\rightarrow}_K t$, with $[r] : \lambda \Rightarrow \rho$ if C, and the term slice $s^\bullet$, we initially assume that all the information introduced by the substitution σ in μ is irrelevant, so that we define the substitution $\theta = \{x/fresh^\bullet \mid x \in Dom(\sigma)\}$ that binds each variable in the domain of σ to a fresh $\bullet$-variable. Then, the algorithm follows a two-step procedure that computes a sequence of substitutions and incrementally refines θ. In both phases, the $\bullet$-symbols in $s^\bullet$ are handled as *existentially quantified variables*, in contrast to the partial stepper of Section 4.2, where $\bullet$-symbols were interpreted to be universally quantified. The first phase retrieves the relevant information contained in the term slice $s^\bullet_{|w}$

of the redex $s_{|w}$, while the second phase recognizes relevant symbols that result from evaluating the rule condition (remind that Maude admits extra-variables that appear in the condition of an equation or rule while they do not appear in the corresponding left-hand side).

Phase 1. This phase first computes the most general unifier between the sliced redex $s^{\bullet}_{|w}$ and the left-hand side λ of the applied rule $[r] : \lambda \Rightarrow \rho$ if C, instantiated with θ, and we compose such unifier with θ and restrict it to the $Dom(\sigma)$'s variables in order to compute ψ_0. This allows the meaningful information of the sliced redex $s^{\bullet}_{|w}$ to be caught while those data that do not appear at meaningful positions are disregarded and not carried on by the rewrite step.

Example 4.4.2

Consider the rewrite rule

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

in Example 0.3.1 together with the following rewrite step $\mathcal{C}$ = $\texttt{next(< 1,1 >,1)} \xrightarrow{\texttt{downN}}_K \texttt{< 1,1+1 >}$ and the term slice $\texttt{next(}\bullet_1\texttt{,1)}$. Let $\theta = \{\texttt{L}/\bullet_2, \texttt{X}/\bullet_3, \texttt{Y}/\bullet_4, \texttt{N}/\bullet_5\}$. In Phase 1, we compute the substitution ψ_0 such that:

$$\begin{aligned}\psi_0 &= (\theta mgu(\texttt{next(}\ \bullet_2 \texttt{<}\ \ \bullet_3\ \texttt{,}\ \bullet_4\ \ \texttt{>,}\ \ \bullet_5\ \texttt{),next(}\bullet_1, \texttt{1)))}_{\restriction\{\texttt{L,X,Y,N}\}} \\ &= (\{\texttt{L}/\bullet_2, \texttt{X}/\bullet_3, \texttt{Y}/\bullet_4, \texttt{N}/\bullet_5\}\{\bullet_1/\ \bullet_2 \texttt{<}\ \ \bullet_3\ \texttt{,}\ \bullet_4\ \ \texttt{>}, \bullet_5/\texttt{1}\})_{\restriction\{\texttt{L,X,Y,N}\}} \\ &= \{\texttt{L}/\bullet_2, \texttt{X}/\bullet_3, \texttt{Y}/\bullet_4, \texttt{N}/\texttt{1}\}\end{aligned}$$

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

Phase 2. This phase detects relevant information that originates from the rule condition. Let $C\sigma = c_1\sigma \wedge \ldots \wedge c_n\sigma$ be the instance of the condition in the rule r that enables the rewrite step μ. We process each (sub)condition $c_i\sigma$, $i = 1, \ldots, n$ by using the auxiliary function *process-condition* given in Figure 4.4 that generates a substitution ψ_i, such that ψ_i is used to further refine the partially ascertained substitution ψ_{i-1} that has been computed by incrementally analyzing the (sub)conditions $c_1\sigma, \ldots, c_{i-1}\sigma$.

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

Now, the term slice $t^{\bullet}$ is computed from $s^{\bullet}$ by replacing its subterm at position w with the instance $(\rho\psi_n)$ of the right-hand side of the applied rule r. This way, all the relevant/irrelevant information detected is transferred into the resulting sliced term $t^{\bullet}$.

Similarly to the case of the backward trace slicing algorithm given in Chapter 1, the *process-condition* function handles matching conditions, rewrite expressions, and equational conditions differently. Specifically, the substitution ψ_i that is returned after processing each condition c_i is computed as follows.

– **Matching conditions.** The analysis of the matching condition $p := m$ during the slicing process of μ is implemented in *process-condition*, as in the backward slicing algorithm, by distinguishing the following two cases.

Case i. If $p\sigma = m\sigma$, then there is no need to generate the canonical form of $m\sigma$, since $p\sigma$ and $m\sigma$ are the same term. Hence, we discover new (possibly) relevant bindings for variables in p by computing the mgu δ between p and $m\psi$. Then, the algorithm returns the parallel composition of δ and ψ (restricted to $Dom(\psi)$'s variables) that updates the input substitution ψ with the new bindings encoded in δ.

Case ii. When $p\sigma \neq m\sigma$, the slicing of the (internal) instrumented execution trace $\mathcal{T}_{int} = m\sigma \rightarrow^{+}_{K} p\sigma$ is required. The slicing process is done by recursively invoking the function $Cslice(m\psi, m\sigma \rightarrow^{+}_{K} p\sigma, \mathcal{I}_{slice})$ whose outcome is the instrumented trace slice $(m\sigma)^{\bullet} \bullet\!\!\rightarrow^{+} (p\sigma)^{\bullet}$ from which new relevant bindings for p's variables can be derived. This is done by computing the mgu δ between p and $(p\sigma)^{\bullet}$; the parallel composition of δ and ψ (restricted to $Dom(\psi)$'s variables) is computed in order to reconciliate ψ with the new bindings in δ.

Note that the analysis above differs from the one of the backward slicing algorithm in which the relevant data come from p instead of m. This is because we slice the internal execution trace

$\mathcal{T}_{int} = m\sigma \to_K^+ p\sigma$ when we proceed forward since we want to transfer the relevant data in $m\sigma$ into $p\sigma$. Also, differently from the backward slicing algorithm, the forward methodology does not deliver a compatibility condition in this phase because it already provides, for free, a means to understand the program behavior w.r.t. user-defined relevant input. Actually, in a forward scenario, the relevant input is known *a priori* and there is no need to infer input restrictions as it happens with backward slicing.

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

– **Equational conditions.** Unlike the evaluation of matching conditions and rewrite expressions, the equational conditions do not generate new bindings for extra-variables during the application of a rewrite step. This means that no new relevant instantiations can be identified for variables that appear in equational conditions. Therefore, in this case, *process-condition* returns the very input substitution ψ.

Example 4.4.3

Consider the Maude specification of Example 0.3.1 and the following rewrite step μ : $\{\texttt{p1,< 4,4 >,1}\}$ `||` $\{\texttt{p2,< 3,5 >,1}\} \overset{\texttt{walk},\sigma_{\texttt{walk}},2}{\to_K} \{\texttt{p1,< 4, 4 >,1}\}$ `||` $\{\texttt{p2,< 3,5 > < 4,5 >,1+1}\}$. Let $\bullet_1$ `||` $\{\bullet_2, \texttt{< 3,5 >}, \bullet_3\}$ be a term slice of $\{\texttt{p1,< 4,4 >,1}\}$ `||` $\{\texttt{p2,< 3,5 >,1}\}$. Since the rewrite step μ occurs at position 2, which is a meaningful position of the term slice $\bullet_1$ `||` $\{\bullet_2, \texttt{< 3,5 >}, \bullet_3\}$, the two-phase procedure described above is applied.

Phase 1. The substitution ψ_0 is computed as follows.

$$\begin{aligned}\psi_0 &= (\theta mgu(\{\bullet_4, \bullet_5, \bullet_6\}, \{\bullet_2, \texttt{< 3,5 >}, \bullet_3\})_{\restriction\{\texttt{PY,L,M,P}\}} \\ &= (\{\texttt{PY}/\bullet_4, \texttt{L}/\bullet_5, \texttt{M}/\bullet_6, \texttt{P}/\bullet_7\}\{\bullet_4/\bullet_2, \bullet_5/\texttt{< 3,5 >}, \bullet_6/\bullet_3\})_{\restriction\{\texttt{PY,L,M,P}\}} \\ &= \{\texttt{PY}/\bullet_2, \texttt{L}/\texttt{< 3,5 >}, \texttt{M}/\bullet_3, \texttt{P}/\bullet_7\}\end{aligned}$$

where $\{\texttt{PY}, \texttt{L}, \texttt{M}\}$ is the left-hand side of the `walk` rule of Example 0.3.1. Note that the variable `L` in ψ_0 is bound to meaningful information, while `PY` and `M` are not considered to be relevant.

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

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

whose instrumented trace slice w.r.t. $\mathcal{I}_{slice}$ and $\texttt{next}(\texttt{L},1)\psi_0$ coincides with $\mathcal{T}_{int}$, since $\texttt{next(L,1)}\psi_0 = \texttt{next(< 3,5 >,1)}$.

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

$$\begin{aligned}\psi_1 &= (\delta \Uparrow \psi_0)_{\restriction\{\texttt{PY,L,M,P}\}}\\ &= (\{\texttt{P/< 4,5 >}\} \Uparrow \{\texttt{PY/}\bullet_2, \texttt{L/< 3,5 >}, \texttt{M/}\bullet_3, \texttt{P/}\bullet_7\})_{\restriction\{\texttt{PY,L,M,P}\}}\\ &= \{\texttt{PY/}\bullet_2, \texttt{L/< 3,5 >}, \texttt{M/}\bullet_3, \texttt{P/< 4,5 >}\}\end{aligned}$$

that updates the information of ψ_0 with the discovered binding $\{\texttt{P/< 4,5 >}\}$ in δ.

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

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

$$\psi_2 = \psi_1 = \{\texttt{PY/}\bullet_2, \texttt{L/< 3,5 >}, \texttt{M/}\bullet_3, \texttt{P/< 4,5 >}\}$$

In symbols,

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

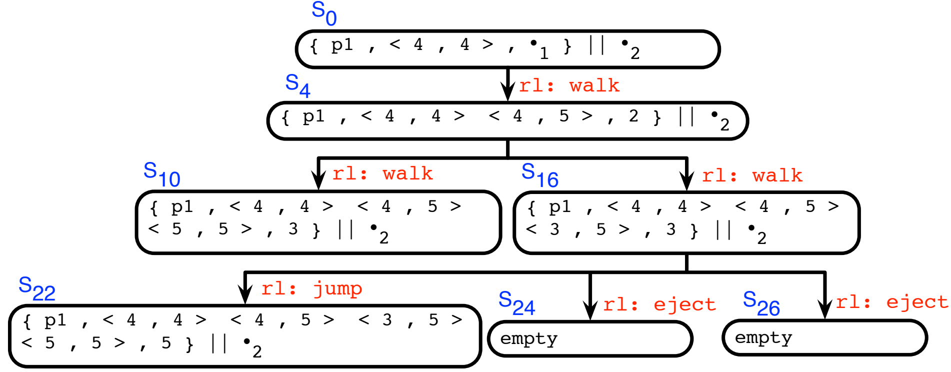


Figure 4.5: Computation tree slice fragment for $s_0^\bullet$ w.r.t. $\mathcal{I}_{slice}$.

The following example describes the interactive construction of a fragment of an instrumented computation tree slice based on the $\mathcal{I}_{slice}$ inspection modality. The example also demonstrates how forward trace slicing can be fruitfully employed to debug RWL specifications. For the sake of readability, in the resulting computation tree slice fragment we omit all instrumentation steps, as in Example 4.2.1.

Example 4.4.4

Consider the computation tree of Figure 3.1 whose initial term is $s_0 = \{\texttt{p1,< 4,4 >,1}\}\ \texttt{||}\ \{\texttt{p2,< 3,5 >,1}\}$. Let $s_0^\bullet = \{\texttt{p1,< 4,4 >,}\bullet_1\}\ \texttt{||}\ \bullet_2$ be a term slice of s_0 where only player `p1` and its corresponding positions are observed. We get the computation tree slice fragment shown in Figure 4.5 by first expanding (w.r.t. the inspection function $\mathcal{I}_{slice}$) the node $s_0^\bullet$ into $s_4^\bullet$, then the node $s_4^\bullet$ into $s_{10}^\bullet$ and $s_{16}^\bullet$, and then the node $s_{16}^\bullet$ into $s_{22}^\bullet$, $s_{24}^\bullet$ and $s_{26}^\bullet$.

The slicing process automatically computes a computation tree slice fragment that represents a partial view of the maze game interactions from player `p1`'s perspective. Actually, irrelevant information is hidden and rules applied on irrelevant positions are directly ignored, which allows a simplified slice to be obtained thus favoring its inspection for debugging and analysis purposes. In fact, by isolating `p1` movements in the tree slice fragment computed by slicing, the user can immediately

observe and debug the program. Specifically, by expanding the term slice $s^\bullet_{16} = \{\texttt{p1,< 4,4 > < 4,5 > < 3,5 >,3}\}$ || $\bullet_2$ into $s^\bullet_{22}$ by an application of the `jump` rule, and expanding $s^\bullet_{16}$ also into $s^\bullet_{24}$ and $s^\bullet_{26}$ by an application of the `eject` rule, the user can immediately realize that the player continues wandering for a while despite being ejected from the game, which clearly reveals the bug in the applied `eject` rule. To prevent `eject` from being nondeterministically applied in competition with other rules such as `walk` or `jump`, `eject` must be programmed as an equation to be deterministically used for normalizing the system state after a player is ejected. Note that the computation tree slice fragment shown in Figure 4.5 cannot be deployed by means of partial stepping since the chosen term slice is overly restrictive to perform a partial rewrite step.

4.5 Forward Trace Slicer Correctness

Forward trace slicing produces instrumented trace slices that are generally not correct in the sense of Definition 4.3.1. This is because $\bullet$ symbols in the $\mathcal{I}_{slice}$ inspection function are interpreted as existential variables, while the partial stepper inspection modality $\mathcal{I}_{pstep}$ handles them as universally quantified variables. Let us see an example.

Example 4.5.1

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

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

Observe that $\mu^\bullet$ is not universally correct according to Definition 4.3.1. Indeed, for every instance of $f(\bullet_1, b)$ that replaces $\bullet_1$ with a term that is different from the constant a, there exists no instrumented trace slice concretization of $\mu^\bullet$.

Nonetheless, every execution trace $\mathcal{T}$ can be always reconstructed from a non-empty instrumented trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ by suitably instanti-

ating all the $\bullet$-variables that appear in $\mathcal{T}^\bullet$. This allows us to formalize the following notion of existential correctness.

Definition 4.5.2 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K \cdots \stackrel{r_n,\sigma_n,w_n}{\rightarrow}_K s_n)$ be an instrumented execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}$ be an inspection function. Then, an instrumented trace slice $s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}$ and $s_0^\bullet$ is* existentially correct iff every s_i is an instance of $s_i^\bullet$, with $i = 0, \ldots, n$.

Example 4.5.3

Consider the instrumented trace slice $\mu^\bullet = (f(\bullet_1, b) \;\; \bullet\!\!\to \;\; b)$ of $f(a,b) \stackrel{r,\{x/b\},\Lambda}{\rightarrow}_K b$ w.r.t. $\mathcal{I}_{slice}$ and $f(\bullet_1, b)$ in Example 4.5.1. Then, $\mu^\bullet$ is existentially correct.

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

Proposition 4.5.4 *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{T} = (s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K \cdots \stackrel{r_n,\sigma_n,w_n}{\rightarrow}_K s_n)$ be an instrumented execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $s_0^\bullet$ be a term slice of s_0. Let $\mathcal{I}_{slice}$ be the forward slicing inspection function. Then, every instrumented trace slice $s_0^\bullet \bullet\!\!\to \cdots \bullet\!\!\to s_n^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{I}_{slice}$ and $s_0^\bullet$ is existentially correct.*

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

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

Case $w_1 \in \mathcal{MP}os(s_0^\bullet)$. By hypothesis, $s_0^\bullet$ is a term slice of s_0, thus s_0 is an instance of $s_0^\bullet$ (in symbols, $s_0^\bullet \leq s_0$). Now,

observe that $s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet$ has been obtained by calling the function $Cslice(s_0^\bullet, s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K s_1, \mathcal{I}_{slice})$. Hence, $\mathcal{T}_1^\bullet = (s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet) = Cslice(s_0^\bullet, s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K s_1, \mathcal{I}_{slice})$. This implies that the following transition $\langle s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K s_1, s_0^\bullet\rangle \Longrightarrow^* \langle \mathtt{nil}, s_0^\bullet \bullet\!\!\rightarrow s_1^\bullet\rangle$ has been performed in the transition system $(Conf, \Longrightarrow)$ by means of one application of the trans rule. By definition of the trans rule (see Figure 3.2), $s_1^\bullet = \mathcal{I}_{slice}(s_0^\bullet, s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K s_1)$. Now, by Definition of $\mathcal{I}_{slice}$, $s_1^\bullet = \mathcal{I}_{slice}(s_0^\bullet, s_0 \stackrel{r_1,\sigma_1,w_1}{\rightarrow}_K s_1) = s_0^\bullet[\rho\psi_m]_{w_1}$, where ψ_m is the substitution obtained by applying the inspection function $\mathcal{I}_{slice}$ w.r.t. the rule $[r_1] : \lambda_1 \Rightarrow \rho_1$ if $c_1 \wedge \ldots \wedge c_m$.

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

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

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

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

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

can show that $s^\bullet_{n-1} \bullet\!\!\to s^\bullet_n$ is an existentially correct instrumented trace slice of $s_{n-1} \stackrel{r_n,\sigma_n,w_n}{\to_K} s_n$ w.r.t. $\mathcal{I}_{slice}$ and $s^\bullet_{n-1}$. Therefore, we also have $s^\bullet_n \leq s_n$, which completes the proof.

■

The results in Proposition 4.5.4 can be directly lifted to (fragments of) instrumented computation tree slices that are generated by means of the forward slicing inspection modality $\mathcal{I}_{slice}$, thereby providing an existential correctness result for the overall forward slicing exploration technique.

Theorem 4.5.5 (existential correctness) *Let $\mathcal{R}$ be a conditional rewrite theory. Let $\mathcal{I}_{slice}$ be the forward slicing inspection function. Let $s^\bullet_0$ be a term slice of term s_0. Let $\mathcal{T}^+_\mathcal{R}(s^\bullet_0)$ be (a fragment of) the instrumented computation tree slice in $\mathcal{R}$ w.r.t. $\mathcal{I}_{slice}$ and $s^\bullet_0$ computed by the function explore$(s_0, s^\bullet_0, \mathcal{R}, \mathcal{I}_{slice})$. Then, each branch in $\mathcal{T}^+_\mathcal{R}(s^\bullet_0)$ is an existentially correct instrumented trace slice w.r.t. $\mathcal{I}_{slice}$ and $s^\bullet_0$ of some instrumented execution trace in $\mathcal{R}$ that originates from s_0.*

Proof. Immediate. It suffices to apply Proposition 4.5.4 to each instrumented Maude step slice in each branch of $\mathcal{T}^+_\mathcal{R}(s^\bullet_0)$. ■

The existential correctness of the computation tree slice $\mathcal{T}_\mathcal{R}(s^\bullet_0)$ naturally derives from the existential correctness of its instrumented counterpart $\mathcal{T}^+_\mathcal{R}(s^\bullet_0)$. In fact, $\mathcal{T}_\mathcal{R}(s^\bullet_0)$ is obtained from $\mathcal{T}^+_\mathcal{R}(s^\bullet_0)$ by hiding all the B-matching transformations and built-in evaluations that occur in the considered instrumentation.

4.6 Backward Trace Slicing as an Instance of the Generic Scheme

Finally, in this section we show how the backward trace slicing technique described in Chapter 1 can be defined as an instance of the generic inspection technique described in Section 3.1.1. This is very convenient as it allows us to reuse large parts of the code that implements Anima.

In the following, we define an inspection function $\mathcal{I}_{bslice}$ that implements the backward trace slicing of a single rewrite step $\mu = (s \stackrel{r,\sigma,w}{\to_K} t)$

and a term slice $t^\bullet$ of t, and delivers the term slice $s^\bullet$ resulting from the back-propagation towards s of the meaningful information in $t^\bullet$.

Let us give a precise formulation of the inspection function $\mathcal{I}_{bslice}$. Given the rule $[r] : \lambda \Rightarrow \rho \text{ } if \text{ } c_1 \wedge \ldots \wedge c_n$ and the rewrite step $\mu = (s \overset{r,\sigma,w}{\rightarrow}_K t)$, let $[r^-] : \rho \Rightarrow \lambda \text{ } if \text{ } c_n^- \wedge \ldots \wedge c_1^-$ be the reverse rule of r where each c_i^- is obtained by reversing the corresponding condition c_i as follows:

$$c_i^- = \begin{cases} m := p & \textbf{if } c_i = (p := m) \\ p \Rightarrow m & \textbf{if } c_i = (m \Rightarrow p) \\ c_i & \textbf{otherwise} \end{cases}$$

Let $\mu^- = (t \overset{r^-,\sigma,w}{\rightleftharpoons}_K s)$ be the reverse step of $\mu = (s \overset{r,\sigma,w}{\rightarrow}_K t)$. Then, we define the inspection function $\mathcal{I}_{bslice}(t^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t) = \mathcal{I}_{slice}(t^\bullet, t \overset{r^-,\sigma,w}{\rightleftharpoons}_K s)$.

Let us see an example.

Example 4.6.1

Consider the conditional rewrite rule

```
crl [r] : f(X,Y) => h(Z,Y) if Z := p(X) /\ Y > 0
```

together with the equation `eq [e] : p(X) = X`, and consider the rewrite step μ : `f(2,5)` $\rightarrow_K$ `h(2,5)` that uses the rule `r`. Let `h(2,`$\bullet_1$`)` be a term slice of `h(2,5)`. Let `crl [r`$^-$`] : h(Z,Y) => f(X,Y) if Y > 0 /\ p(X) := Z` be the reverse rule of `r`. Then, we can backward slice the step μ by computing $\mathcal{I}_{bslice}($`h(2,`$\bullet_1$`)`$,$ `f(2,5)` $\overset{r,\sigma,w}{\rightarrow}_K$ `h(2,5)`$) = \mathcal{I}_{slice}($`h(2,`$\bullet_1$`)`$,$ `h(2,5)` $\overset{r^-,\sigma,w}{\rightleftharpoons}_K$ `f(2,5)`$) =$ `f(2,`$\bullet_1$`)`.

The reverse version $\mathcal{T}^-$ of an instrumented trace $\mathcal{T}$ can be constructed by reversing each rewrite step in $\mathcal{T}$ as follows.

Given the instrumented trace $\mathcal{T} = (s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow}_K \cdots \overset{r_n,\sigma_n,w_n}{\rightarrow}_K s_n)$, its reversed counterpart is $\mathcal{T}^- = (s_n \overset{r_n^-,\sigma_n,w_n}{\rightleftharpoons}_K \cdots \overset{r_1^-,\sigma_1,w_1}{\rightleftharpoons}_K s_0)$.

Now, the backward slicing process of an instrumented trace can be simply achieved by invoking the function *Cslice* of Definition 3.1.3 with $\mathit{Cslice}(s_n^\bullet, \mathcal{T}^-, \mathcal{I}_{bslice})$.

Example 4.6.2

Consider the rewrite rule and the equation of Example 4.6.1. Given the instrumented trace $\mathcal{T} = \texttt{f(2,5)} \stackrel{\texttt{r},\sigma,w}{\rightarrow}_K \texttt{h(2,5)}$ and its reversed counterpart $\mathcal{T}^{-} = \texttt{h(2,5)} \stackrel{\texttt{r}^{-},\sigma,w}{\rightleftharpoons}_K \texttt{f(2,5)}$, we compute the trace slice $\mathcal{T}^{\bullet} = \mathit{Cslice}(\texttt{h(2,}\bullet_1\texttt{)}, \texttt{h(2,5)} \stackrel{\texttt{r}^{-},\sigma,w}{\rightleftharpoons}_K \texttt{f(2,5)}, \mathcal{I}_{\mathit{bslice}})$ of $\mathcal{T}$ w.r.t. $\texttt{h(2,}\bullet_1\texttt{)}$ using $\mathcal{I}_{\mathit{bslice}}$ as follows.

The transition $\langle \texttt{h(2,5)} \stackrel{\texttt{r}^{-},\sigma,w}{\rightleftharpoons}_K \texttt{f(2,5)}, \texttt{h(2,}\bullet_1\texttt{)}\rangle \Longrightarrow \langle \texttt{nil}, \mathcal{T}^{\bullet}\rangle$ is performed in the transition system $(\mathit{Conf}, \Longrightarrow)$ by means of one application of the trans rule of Figure 3.2 that involves a call to the $\mathcal{I}_{\mathit{slice}}$ function given in Figure 4.3, that is, $\mathcal{I}_{\mathit{bslice}}(\texttt{h(2,}\bullet_1\texttt{)}, \texttt{f(2,5)} \stackrel{\texttt{r},\sigma,w}{\rightarrow}_K \texttt{h(2,5)}) = \mathcal{I}_{\mathit{slice}}(\texttt{h(2,}\bullet_1\texttt{)}, \texttt{h(2,5)} \stackrel{\texttt{r}^{-},\sigma,w}{\rightleftharpoons}_K \texttt{f(2,5)})$

Since the reverse rewrite step $\texttt{h(2,5)} \stackrel{\texttt{r}^{-},\sigma,w}{\rightleftharpoons}_K \texttt{f(2,5)}$ occurs at position Λ, which is a meaningful position of the term slice $\texttt{h(2,}\bullet_1\texttt{)}$, then $\theta = \{\texttt{X}/\bullet_2, \texttt{Y}/\bullet_3, \texttt{Z}/\bullet_4\}$ and the two-phase procedure described in Section 4.4 is applied.

Phase 1. The substitution ψ_0 is computed as follows.

$$\begin{aligned}\psi_0 &= (\theta\, mgu(\texttt{h}(\bullet_4,\bullet_3), \texttt{h}(2,\bullet_1))_{\restriction\{\texttt{X,Y,Z}\}} \\ &= (\{\texttt{X}/\bullet_2, \texttt{Y}/\bullet_3, \texttt{Z}/\bullet_4\}\{\bullet_4/2, \bullet_3/\bullet_1\})_{\restriction\{\texttt{X,Y,Z}\}} \\ &= \{\texttt{X}/\bullet_2, \texttt{Y}/\bullet_1, \texttt{Z}/2\}\end{aligned}$$

Phase 2. First the condition $\texttt{Y} > \texttt{0}$ is processed. Since it is an equational condition, the call to the *process-condition* function will return the substitution ψ_1 such that $\psi_1 = \psi_0$, which implies that no new relevant information has been identified.

Subsequently, the inverted condition $\texttt{p(X)} := \texttt{Z}$ is processed by calling the function *process-condition*$(\texttt{p(X)} := \texttt{Z}, \sigma, \{\texttt{X}/\bullet_2, \texttt{Y}/\bullet_1, \texttt{Z}/2\})$. In this specific case, we have to consider the internal execution trace

$$\mathcal{T}_{int} = 2 \stackrel{\texttt{e}}{\rightarrow}_K \texttt{p(2)}$$

whose instrumented trace slice w.r.t. $\mathcal{I}_{\mathit{slice}}$ and $\texttt{Z}\psi_1$ coincides with $\mathcal{T}_{int}$, that is:

$$\mathcal{T}^{\bullet}_{int} = 2 \bullet\!\!\rightarrow \texttt{p(2)}$$

Hence, $\delta = mgu(\texttt{p(X)}, \texttt{p(2)}) = \{\texttt{X}/2\}$. The evaluation of the condition `p(X) := Z` terminates by returning the substitution $\psi_2 = \{\texttt{X}/2, \texttt{Y}/\bullet_1, \texttt{Z}/2\}$.

Finally, the term slice of `f(2,5)` is computed by replacing the subterm at position Λ of the term slice $\texttt{h(2,}\bullet_1\texttt{)}$ with the instance $\texttt{f(X,Y)}\psi_2$, which is $\texttt{f(2,}\bullet_1\texttt{)}$.

Thus, the computed trace slice is $\mathcal{T}^\bullet = \texttt{f(2,}\bullet_1\texttt{)} \bullet\!\!\to \texttt{h(2,}\bullet_1\texttt{)}$.

Note that, if we would apply the computed substitution ψ_2 to each equational condition of `r`, that is $(\texttt{Y > 0})\psi_2 = \bullet_1 > 0$, we also get the backward trail $[\texttt{f(2,}\bullet_1\texttt{)} \bullet\!\!\to \texttt{h(2,}\bullet_1\texttt{)}, \bullet_1 > 0]$, which is correct w.r.t. the notion of correctness given in Section 1.3.

Chapter 5

The Anima system

The exploration methodology developed in Part II of the thesis has been implemented in the Anima tool, which is publicly available at [Ani14] is written in Maude and C++ and consists of about 270 Maude function definitions (approximately 2500 lines of source code) together with the implementation in C++ of `metaReducePath`, a new Maude command provided by our implementation that is described in Section 5.2. Anima also comes with an intuitive Web user interface based on AJAX technology, which allows users to graphically animate their programs and display fragments of computation trees. The core exploration engine is specified as a RESTful Web service by means of the Jersey JAX-RS API. The architecture of Anima is depicted in Figure 5.1 and consists of five main components: Anima Client, JAX-RS API, Anima Web Service, Database, and Anima Core. The Anima Client is purely implemented in HTML5 Canvas[1] and JavaScript. It represents the front-end layer of our tool and provides an intuitive, versatile Web user interface, which interacts with the Anima Web Service to invoke the capabilities of the Anima Core and save partial results in the MongoDB Database component, which is a scalable, high-performance, open source NoSQL database that perfectly fits our needs.

5.1 The Anima Exploration Tool

The main features of Anima include the following:

1. *File uploading.* Maude specifications can be uploaded in Anima either as a simple *.maude* file or as a compressed *.zip* file, which

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

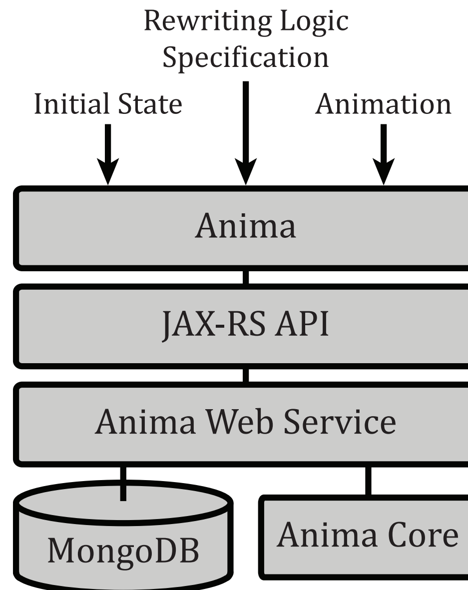


Figure 5.1: Anima architecture.

must contain all the required files for the specification to work properly.

2. *Inspection strategies.* The tool implements the three inspection strategies described in Section 4. As shown in Figure 5.2, the user can select the desired strategy by using the selector provided in the option pane.

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

4. *Expansion/Folding of program states.* The user can expand or fold states of the tree by left-clicking with the mouse on their state label, or by right-clicking with the mouse on the node and then selecting either the *Expand Node* option, the *Expand Subtree* option, or the *Fold Node* option that are offered in the contextual menu. The

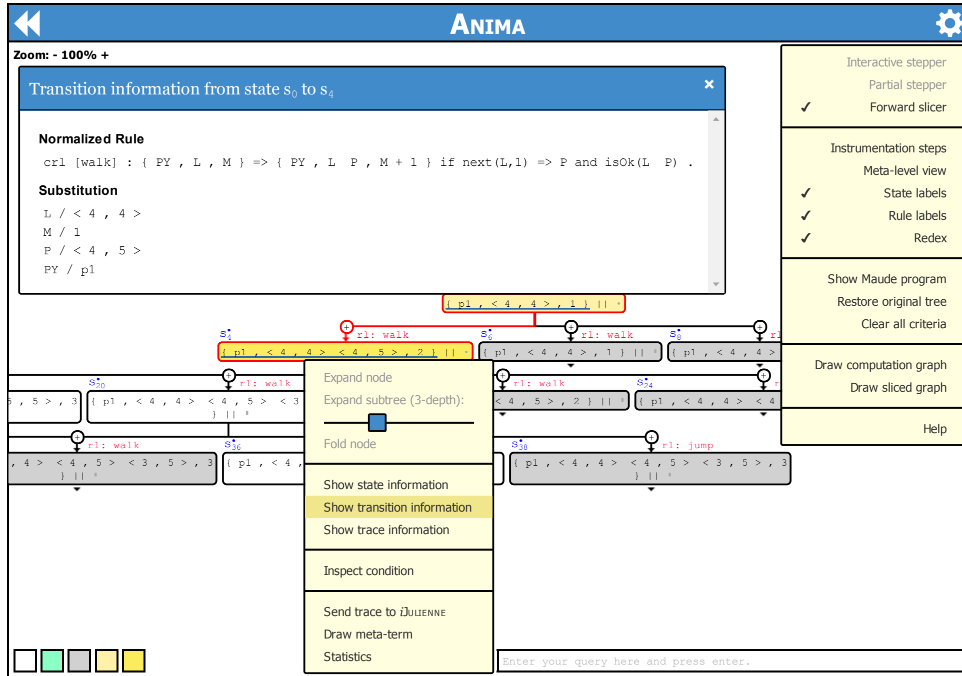


Figure 5.2: Anima at work.

Expand Subtree option allows the user to automatically expand, up to a given depth k, for $k \leq 5$ (with default depth $k = 3$, which can be tuned by means of a slider), the subtree hanging from the considered node by following a breadth-first strategy.

When a state slice that is situated at the frontier of the computed tree slice fragment is selected for Expansion/Folding, the whole branch leading from the root of the tree to the selected node is highlighted, as illustrated in Figure 5.2. Common actions like dragging, zooming, and navigating the tree are allowed. Also, when a tree node is selected, the position of the tree on the screen is automatically rearranged to keep the chosen node at the center of the scene.

5. *Display of instrumented steps.* The user can freely choose to display either a default, simplified view of a rewrite step (where only the applied rewrite rule is displayed), or the complete and detailed

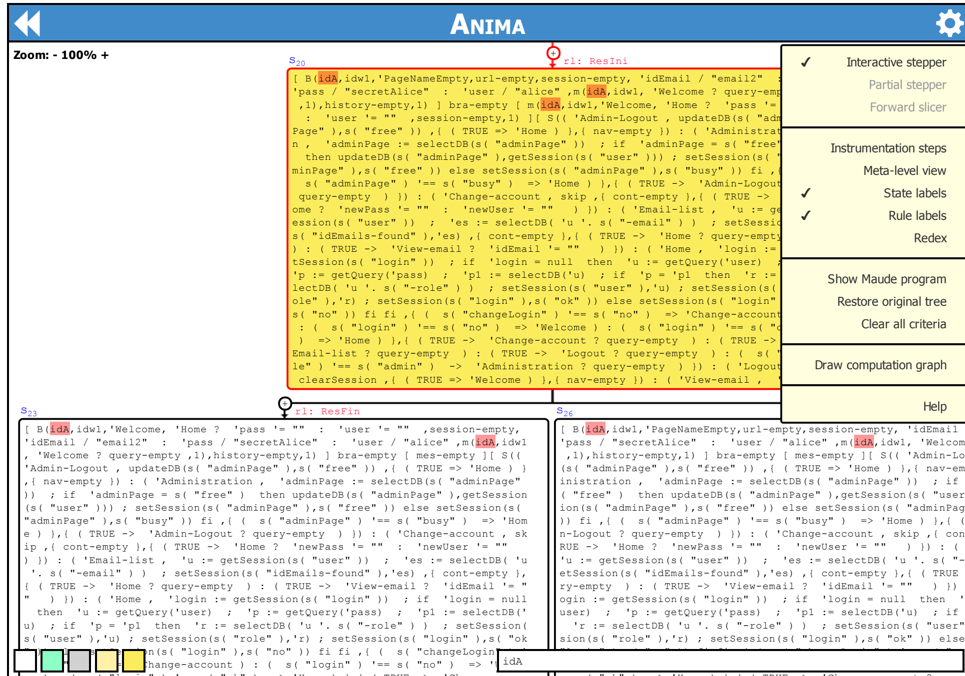


Figure 5.3: Anima search mechanism.

sequence of steps in the corresponding instrumented trace that simulates the step. This facility can be locally accessed by clicking on the +/− symbols that respectively adorn the standard/instrumented view of the rewrite step, or by checking/unchecking the *Instrumented steps* option in the Anima option pane for the entire computation tree.

6. *Tree Query mechanism.* The search facility illustrated in Figure 5.3 implements a pattern language that allows the selected information of interest to be searched in huge states of complex computation trees. The user only has to provide a filtering pattern (the query) that specifies the set of symbols that he/she wants to search for, and then all the states matching the query are automatically highlighted in the computation tree.

7. *Showing rewrite step information.* Anima facilitates the inspection of any rewrite step $s \to t$ of the computation tree by underlining

Trace information

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

Figure 5.4: Anima trace information.

the differences between the two states (typically the selected redex of s and its contractum in t). In the case of a non-instrumented step $s \rightarrow_{\Delta,B} t$ (resp. $s \rightarrow_{R,B} t$), we generally cannot highlight the redex and contractum of the step as they might not exist in s and t because of the matching modulo B that precedes the rewrite step and the normalization that occurs after the rewrite step. Actually, recall that s and t are eventually reordered, augmented with unity elements, and parenthesised, yielding the B-equivalent terms s' and t' that star in an intermediate rewrite step $s' \rightarrow_{\Delta} t'$ (resp., $s' \rightarrow_R t'$). In this case, we underline the antecedents in s of the reduced redex in s' (and the descendants in t of the contractum that appears in t').

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

8. *Showing trace information.* By right-clicking a tree node and by selecting the *Show trace information* option, the user can obtain the complete information of the execution trace from the root to the selected node. This information is presented in a table that includes the labels of the rules and equations applied, the terms that result from the application of each rule or equation and the computed

trace slice (if applicable) as shown in Figure 5.4. Moreover, Anima offers the possibility to export the displayed trace into meta-level representation, so the user can easily transfer the selected trace to any other Maude trace analyzer tool like the online backward trace analysis tool *i*Julienne [ABFS13b], for example.

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

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

11. *Forward-Backward slicing integration.* In order to facilitate the exhaustive and incremental inspection of a given trace, Anima offers the possibility to export the trace to *i*Julienne [ABFS13b], which allows the origins or antecedents of a given expression (that is, those symbols in the initial state from which the observed expression descends) to be identified. This is done by tracing back all control and data dependencies.

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

12. *Inspection of conditions.* As shown in Figure 5.5, Anima facilitates the inspection of the conditions satisfied during the application of a conditional rule or equation by right-clicking on the generated state and then selecting the *Inspect condition* option, which allows the

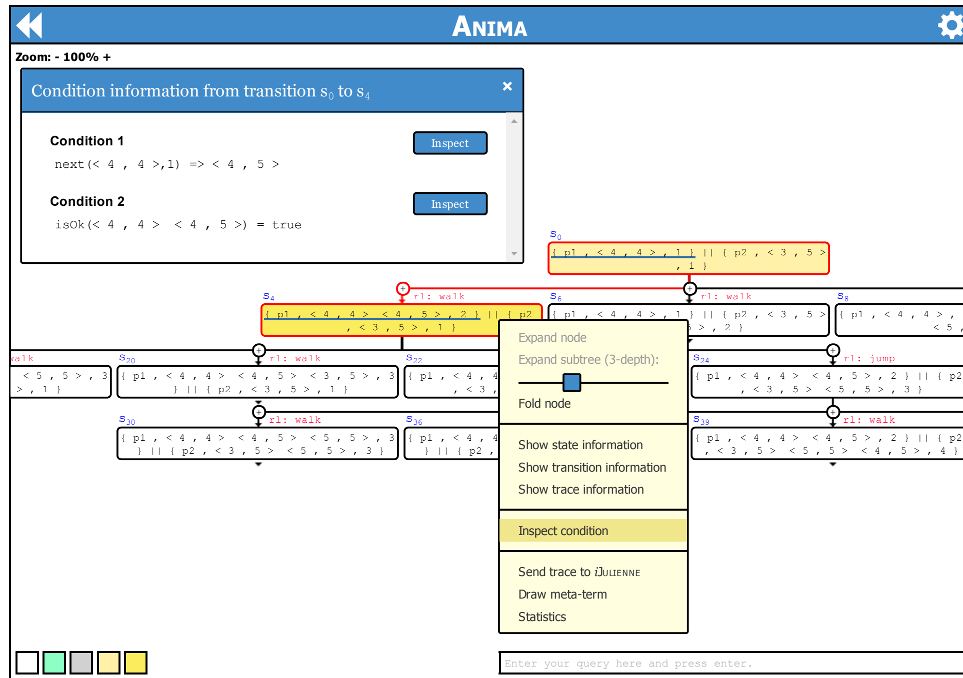


Figure 5.5: Inspection of a condition with Anima.

user to export the traces deployed by evaluating the rule conditions to *i*JULIENNE for further analysis.

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

5.2 Implementation of the Tool

One of the main challenges in the implementation of a trace-based Maude tool such as Anima is to make explicit the concrete sequence of internal term transformations occurring in a particular Maude execution trace, which is generally hidden and inaccessible within Maude's rewriting ma-

chinery. For the case of rule applications, this sequence can be easily retrieved by means of the Maude `metaSearchPath` command, but a similar command does not exist to ascertain the sequence of built-in operators and equations applied. These are only recorded in a raw text output trace, which cannot be manipulated as a meta-level expression by Maude. In order to overcome this drawback, we have implemented our own Maude command, named `metaReducePath`, which returns the detailed sequence of transformations (using equations, built-in operators, and any internal normalizations) applied to a term until its canonical form is reached.

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

```
sort ITrace ITraceStep .
subsort ITraceStep < ITrace .

op nil : -> ITrace [ctor] .
op __ : ITrace ITrace -> ITrace [ctor assoc id: nil] .
op {_,_,_} : Equation Substitution Context
             -> ITraceStep [ctor] .
op metaReducePath : Module Term ~> ITrace [special (...)] .
```

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

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

More specifically, given a reduction step $s \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,B} t$, these selectors respectively return: (i) the equation e, (ii) the substitution σ, and (iii)

the *context*[2] surrounding the redex and in which the replacement takes place.

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

Example 5.2.1

Observe the reduction of the input term `g(f(a, b), f(a, b), f(a, b))` in the following functional module:

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

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

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

These unorthodox *multi-steps* are the side effect of two efficient implementation optimizations, namely the alliance of identical subterms using multiplicity superscripts (as in $f(\alpha^2, \beta^3, \gamma)$) [Eke03] and *in-place rewriting* [Eke14], which means that equational rewriting destroys the DAG structure representing the term that is rewritten at each rewrite step. This is great for efficiency, but it is a major obstacle for exploring the computations. The new command `metaReducePath` provided by Anima detects and standardizes the *multi-steps* at the meta-level by spreading them out into the necessary number of single reduction steps, one for each single equation application, while explicitly recording the associated context information.

Example 5.2.2

For the specification and input term of Example 5.2.1, the standardized reduction trace that is obtained by invoking the command `metaReducePath` is as follows.[3]

```
Maude> reduce in META-LEVEL :
  metaReducePath(upModule('EXAMPLE, false),
    'g['f['a.Elem,'b.Elem],'f['a.Elem,'b.Elem],
    'f['a.Elem,'b.Elem]]) .

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

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

```
{eq 'f['X:Elem,'Y:Elem] = 'c.Elem [label('EQ1)] .,
  'X:Elem <- 'a.Elem ; 'Y:Elem <- 'b.Elem,
  'g['c.Elem,'c.Elem,[]]}
```

Technically, the execution of `metaReducePath` can be split into two phases: equational simplification and lifting to the meta-level. In the simplification phase, the input term is reduced to canonical form by using Maude's equational simplification. For each applied equation and internal normalization transformation, our command additionally collects all the relevant information that we need to subsequently reconstruct the performed steps. This includes not only built-in evaluation but also memoization and other internal transformations such as the aforementioned `iter`, which replaces chains of iterations of a unary operator by a single instance of the iterated function, raised to the number of iterations, e.g., s(s(s(0))) as $\texttt{s}^3(\texttt{0})$. Once the term has reached its canonical form, the lifting phase consists of raising to the meta-level all the collected information and assembling the resulting instrumented execution traces.

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

Table 5.1: Execution results of the `metaReducePath` command for `fibo(n)`.

Table 5.1 provides some figures regarding the execution of the `metaReducePath` command. We tested our command on a 3.3GHz Intel Xeon E5-1660 with 64GB of RAM by reducing different calls to the `fibo`

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

Figure 5.6: Benchmark problem for the `metaReducePath` command.

function given in Figure 5.6. In Table 5.1, we distinguish the two phases mentioned above, namely equational simplification and lifting. For the equational simplification phase, the number of rewrites and the reduction times are given. For the lifting phase, we show the problem size, the length of the resulting instrumented execution traces, and the processing times. The problem size (column ♮ size) is measured as the number of expressions (applied equation, substitution, and context for each step) that are manipulated. The length of the resulting instrumented execution traces (column $|\mathcal{T}|$) is measured as the number of rewrite steps. Note that for extremely huge execution traces such as the trace of `fibo(25)`, which consists of 439,201 rewrite steps, the number of manipulated terms can be very high (more than 1,300,000) yet the execution time is reasonable (a few seconds) and comparable to existing Maude meta-commands that process millions of terms [Eke03].

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

Conclusions

Several methodologies for program comprehension and debugging that rely on the dynamic analysis of execution traces have been developed ever since the pioneers of software development first realized that it is much more difficult to avoid programming errors than desired. However, most of these techniques have to deal with a huge amount of information that is much greater than the one strictly needed to identify and reproduce the source of error. One may think that the more information we manage about a problem, the better; however, when the amount of information exceeds a certain limit, it becomes a major drawback since it may obscure the concrete information that causes the error. The consequences of this excess of information may vary from simple annoyance to the far more troubling inability to perform the required code revision and correction.

In this thesis, we have developed a general scheme that hopefully contributes a step forward in the analysis, comprehension and debugging of concurrent programs.

In Part I, we have presented an incremental, slicing-based backward trace analysis technique for rewriting logic programs that are written and executed in the Maude system. This methodology can drastically reduce the size and complexity of the traces under examination and is useful for the analysis of execution traces of sophisticated rewrite theories that may include rules, equations and equational axioms such as commutativity, associativity, and unity. The technique has been implemented in the *i*JULIENNE system and our experiments reveal that our methodology does not come at the expense of performance with respect to existing Maude tools. This makes *i*JULIENNE attractive for Maude users, especially taking into account that program debugging and trace analysis in Maude is not easy with current state-of-the-art tools. The implementation comprises a front-end that consists of a web graphical user interface and a back-end written in Maude that exploits its meta-level capabilities. The tool can be tuned to reveal all relevant information (including applied equation/rule, redex position, and matching substitution) for each single rewrite step that is obtained by (recursively) applying a rule, equation, or algebraic axiom, which greatly improves the standard view of execu-

tion traces and their meta-representations in Maude. In particular, it can provide both, a textual representation of the trace and its meta-level representation to be used for further automated analyses, including the *incremental* trace slicing algorithm that supports stepwise refinements of the trace slice.

In Part II, we have formalized a rich and highly dynamic, parameterized scheme for the trace inspection of conditional rewrite theories based on a generic animation algorithm that can be tuned to work with different modalities, including *incremental stepping*, *partial stepping* and *automated forward/backward slicing*, which drastically reduces the size and complexity of the computations under examination. The algorithm is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates conditional RWL theories that involve rewriting modulo associativity, commutativity, and unity axioms. The proposed methodology is implemented and tested in the graphical tool Anima, which provides a skillful and highly dynamic interface for the dynamic analysis of RWL computations. The tool is useful for Maude programmers in two ways. First, it graphically exemplifies the semantics of the language, allowing the evaluation rules to be observed in action. Secondly, it can be used as a debugging tool, allowing the users to step forward and backward while slicing the trace in order to validate input data or locate programming mistakes.

As already mentioned, the present version of Anima supports the instrumentation of matching modulo associativity, commutativity, and (left-, right- or two-sided) unity. In addition, Anima has an extensible design so that instrumentation for other equational axioms such as idempotency can be easily added in the future. As future work, we are interested to extend our exploration technique to more sophisticated rewrite theories that may include membership axioms. We also plan to exploit the dynamic dependencies exposed by our conditional trace slicing methodology to endow Anima with a program slicing capability that can identify those parts of a Maude theory that can (potentially) affect the values computed at some point of interest [Tip95, FT94].

As another line for future work, we also intend to explore the application of our trace slicing methodology to universal debugging and runtime verification [BFF+10], which are concerned with the monitoring, debugging and analysis of system executions. More specifically, we

can consider a programming language defined in $\mathbb{K}$ [RS10] (e.g., C, Java, JavaScript, etc.) which is a rewriting-based executable semantic framework in which programming languages syntax and semantics can be defined. $\mathbb{K}$ semantics definitions are mechanized using a rewriting engine such as $\mathbb{K}$-Maude [SR10] or SMT solvers [SCM$^+$14]. Hence trace slicing can be used to analyze such semantics-driven executions w.r.t. a reference specification that monitors critical data. This way, debugging and runtime verification might be semantically grounded in our setting, while it is commonly off-hacked in more traditional approaches by using dedicated techniques such as program instrumentation.

Finally, the Maude system currently supports a declarative debugging technique [RVMO10a] *à la* Shapiro [Sha82]. We think that our trace slicing methodology can provide a complementary source of information to shorten and simplify the declarative debugging process. Indeed, by not considering some computations that were proven by the trace slicer to have no influence on a criterion of interest, we might avoid many unnecessary debugging questions to the user.

Bibliography

[ABBF10] M. Alpuente, D. Ballis, M. Baggi, and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In *Proceedings of the 19th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010)*, pages 43–52. Association for Computing Machinery, 2010.

[ABE+11] M. Alpuente, D. Ballis, J. Espert, F. Frechina, and D. Romero. Debugging of Web Applications with WEB-TLR. In *Proceedings of the 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 *Proceedings of the 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 *Proceedings of the 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *Lecture Notes in Computer Science (LNCS)*, pages 34–48. Springer-Verlag, 2011.

[ABF+13] M. Alpuente, D. Ballis, M. Falaschi, F. Frechina, and D. Romero. Rewriting-based Repairing Strategies for XML Repositories. *The Journal of Logic and Algebraic Programming*, 82(8):326–352, 2013.

[ABFR06] M. Alpuente, D. Ballis, M. Falaschi, and D. Romero. A Semi-Automatic Methodology for Repairing Faulty Web Sites. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*, pages 31–40. IEEE Computer Society Press, 2006.

[ABFR12a] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Backward Trace Slicing for Conditional Rewrite Theories. In *Proceedings 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 *Proceedings 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.

[ABFR14] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Using Conditional Trace Slicing for improving Maude Programs. *Science of Computer Programming*, 80, Part B:385 – 415, 2014.

[ABFS13a] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Parametric Exploration of Rewriting Logic Computations. In *Proceedings 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 *i*JULIENNE. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science (LNCS)*, pages 121–124. Springer-Verlag, 2013.

[ABFS14] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way). In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014)*, volume 8373 of *Lecture Notes in Computer Science (LNCS)*, pages 229–255. Springer-Verlag, 2014.

[ABFS15] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation*, 2015. http://dx.doi.org/10.1016/j.jsc.2014.09.028.

[ABR09] M. Alpuente, D. Ballis, and D. Romero. Specification and Verification of Web Applications in Rewriting Logic. In *Proceedings 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.

[ABR14] M. Alpuente, D. Ballis, and D. Romero. A Rewriting Logic Approach to the Formal Specification and Verification of Web Applications. *Science of Computer Programming*, 81:79–107, 2014.

[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.

[ALL96] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and Caching of Dependencies. *ACM SIGPLAN Notices*, 31(6):83–91, 1996.

[Ani14] The Anima Web site, 2014. Available at: http://safe-tools.dsic.upv.es/anima.

[BBF09] M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In *Proceedings 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. Roşu, O. Sokolsky, and N. Tillmann, editors. *Proceedings of the 1st International Conference on Runtime Verification (RV 2010)*, volume 6418 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2010.

[BKV00] I. Bethke, J. W. Klop, and R. de Vrijer. Descendants and Origins in Term Rewriting. *Information and Computation*, 159(1–2):59–124, 2000.

[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 *Proceedings 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, 2012.

[CAA11] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as Dependency Analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*. Springer-Verlag, 2007.

[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/`.

[CFF01] J. Clements, M. Flatt, and M. Felleisen. Modeling an Algebraic Stepper. In *Proceedings of the 10th European Symposium on Programming (ESOP 2001)*, volume 2028 of *Lecture Notes in Computer Science (LNCS)*, pages 320–334. Springer-Verlag, 2001.

[CR09] F. Chen and G. Roşu. Parametric Trace Slicing and Monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for 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 *Proceedings 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.

[DKT93] A. Van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15(5–6):523–545, 1993.

[DM97] I. Durand and A. Middeldorp. Decidable Call by Need Computations in Term Rewriting (Extended Abstract). In *Proceedings of the 14th International Conference on Automated Deduction (CADE 1997)*, volume 1249 of *Lecture Notes in Computer Science (LNCS)*, pages 4–18. Springer-Verlag, 1997.

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

[Duc99] M. Ducassé. OPIUM: An Extendable Trace Analyzer for PROLOG. *The Journal of Logic Programming*, 39(1–3):177–223, 1999.

[Eke95] S. Eker. Associative-Commutative Matching via Bipartite Graph Matching. *The Computer Journal*, 38(5):381–399, 1995.

[Eke03] S. Eker. Associative-Commutative Rewriting on Large Terms. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science (LNCS)*, pages 14–29. Springer-Verlag, 2003.

[Eke14] S. Eker, 2014. Personal Communication.

[EMM06] S. Escobar, C. Meadows, and J. Meseguer. A Rewriting-Based Inference System for the NRL Protocol Analyzer and its Meta-Logical Properties. *Theoretical Computer Science (TCS)*, 367(1):162–202, 2006.

[EMMS14] S. Escobar, C. Meadows, J. Meseguer, and S. Santiago. A rewriting-based forwards semantics for Maude-NPA. In *proc. HotSoS*, 2014. To appear.

[Fay79] M. Fay. First Order Unification in an Equational Theory. In *Proceedings of the 4th International Conference on Automated Deduction (CADE 1979)*, pages 161–167. Academic Press, Inc., 1979.

[FR01] M. A. Francel and S. Rugaber. The Value of Slicing While Debugging. *Science of Computer Programming*, 40(2–3):151–169, 2001.

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

[HL79] G. Huet and J.-J. Lévy. Call by need Computations in Nonambiguous Linear Term Rewriting Systems. Technical Report 359, INRIA, 1979.

[HL91] G. P. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems, I. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–414. The MIT Press, 1991.

[Hof11] P. Hofstedt. *Multiparadigm Constraint Programming Languages*. Springer-Verlag, 2011.

[HR01] K. Havelund and G. Roşu. Java PathExplorer - A Runtime Verification Tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2001): A New Space Odyssey*. Canadian Space Agency, 2001.

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

[jsl08] Jslice: a Dynamic Slicing Tool for Java Programs, 2008. Available at: `http://jslice.sourceforge.net`.

[KL88] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[Klo90] J.W. Klop. Term Rewriting Systems. Technical Report CS-R9073, Centre for Mathematics and Computer Science, 1990.

[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.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1996)*, volume

1055 of *Lecture Notes in Computer Science (LNCS)*, pages 147–166. Springer-Verlag, 1996.

[LS03] Y. A. Liu and S. D. Stoller. Eliminating Dead Code on Recursive Data. *Science of Computer Programming*, 47(2–3):221–242, 2003.

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

[Mes08] J. Meseguer. The Temporal Logic of Rewriting: A Gentle Introduction. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of his 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science (LNCS)*, pages 354–382. Springer-Verlag, 2008.

[Mid99] A. Middeldorp. Strategies for Rewrite Systems : Normalization and Optimality. *Kôkyûroku Bessatsu - Languages, Algebra and Computer Systems*, 1106:149–160, 1999.

[MOM02] N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science (TCS)*, 285(2):121–154, 2002.

[MOPV12] N. Martí-Oliet, M. Palomino, and A. Verdejo. Rewriting Logic Bibliography by Topic: 1990ï¿½2011. *The Journal of Logic and Algebraic Programming*, 81(7–8):782–815, 2012.

[O'D77] M. J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1977.

[Pal90] C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, volume 443 of *Lecture Notes in Computer Science (LNCS)*, pages 386–399. Springer-Verlag, 1990.

[Plo04] G. D. Plotkin. The Origins of Structural Operational Semantics. *The Journal of Logic and Algebraic Programming*, 60–61(1):3–15, 2004.

[RAA13] A. Riesco, I. Mariuca Asavoae, and M. Asavoae. A generic program slicing technique based on language definitions. In *Proceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012)*, volume 7841 of *Lecture Notes in Computer Science (LNCS)*, pages 248–264. Springer-Verlag, 2013.

[RB05] N. F. Rodrigues and L. S. Barbosa. Component Identification Through Program Slicing. In *Proceedings of the 2nd International Workshop on Formal Aspects of Component Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 291–304. Elsevier Science, 2005.

[RS10] G. Roşu and T. F. Serbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[RVCMO09] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative Debugging of Rewriting Logic Specifications. In *Proceedings of the 19th International Workshop on Algebraic Development Techniques (WADT 2008)*, volume 5486 of *Lecture Notes in Computer Science (LNCS)*, pages 308–325. Springer-Verlag, 2009.

[RVMO10a] A. Riesco, A. Verdejo, and N. Martí-Oliet. A Complete Declarative Debugger for Maude. In *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology (AMAST 2010)*, volume 6486 of *Lecture Notes in Computer Science (LNCS)*, pages 216–225. Springer-Verlag, 2010.

[RVMO10b] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative Debugging of Missing Answers for Maude. In *Proceedings*

of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010), volume 6 of *LIPIcs*, pages 277–294. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

[RVMOC12] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *The Journal of Logic and Algebraic Programming*, 81(7–8):851–897, 2012.

[SCM+14] A. Stefănescu, S. Ciobâcă, R. Mereuta, B. Moore, T. F. Serbănută, and G. Roşu. All-path reachability logic. In *Rewriting and Typed Lambda Calculi*, volume 8560 of *Lecture Notes in Computer Science*, pages 425–440. Springer International Publishing, 2014.

[Sha82] E. Y. Shapiro. Algorithmic Program Diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982)*, pages 299–308. Association for Computing Machinery, 1982.

[Sla74] J.R Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM (JACM)*, 21(4):622–642, 1974.

[SM03] A. Sabelfeld and A.C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SR10] T. F. Serbănuţă and G. Roşu. K-maude: A rewriting based tool for semantics of programming languages. In *Rewriting Logic and Its Applications*, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122. Springer Berlin Heidelberg, 2010.

[Tal08] C. Talcott. Pathway Logic. In *Proceedings of the 8th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2008)*, volume 5016 of *Lecture Notes in Computer Science (LNCS)*, pages 21–53. Springer-Verlag, 2008.

[TeR03] TeReSe. *Term Rewriting Systems.* Cambridge University Press, 2003.

[Tip95] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[VO01] G. Villavicencio and J. N. Oliveira. Reverse Program Calculation Supported by Code Slicing. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 35–46. IEEE Computer Society Press, 2001.

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

Appendix A

Maude Specification of the Experimental Evaluation Examples

This appendix contains the Maude specification of the experimental evaluation examples discussed in Chapter 2.

Fault-Tolerant Client-Server Communication Protocol

The formal specification of the fault-tolerant client-server communication protocol example is as follows:

```
mod CLIENT-SERVER-TRANSF is

    inc NAT .

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

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

    ops Srv-A Srv-B : -> ServName .
    ops Cli-A Cli-B : -> CliName .
    op null : -> State .
    op _&_ : State State -> State [assoc comm id: null] .
```

```
    op _<-_ : Host Content -> Msg .
    op {_,_} : Host Data -> Content .
    op [_,_,_,_] : CliName ServName Question Answer -> Cli .
    op na : -> Answer .
    op [_] : ServName -> Serv .
    op f : ServName CliName Question -> Answer .

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

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

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

endm
```

Needham-Schroeder Network Authentication Protocol

Regarding the Needham-Schroeder network authentication protocol example, its formal specification (based on [EMMS14]) is as follows:

```
mod NS is

  sort Universal . --- Special sort used for unsorted actions
  sort Msg . --- Generic sort for messages
```

```
sort Fresh . --- Sort for private information.
sort Public . --- Handy sort to say what is public

subsort Public < Msg .

op emptyPublic : -> Public .
op nullFresh : -> Fresh .

sort MsgSet .

subsort Msg < MsgSet .

op emptyMsgSet : -> MsgSet [ctor] .
op _`,_ : MsgSet MsgSet -> MsgSet [ctor assoc comm
                                    id: emptyMsgSet] .
op noMsg : -> Msg . --- Auxiliar useless message
                        used as a marker

sort SMsg .
sort SignedSMsg .

subsort SignedSMsg < SMsg .

op +`(_`) : Msg -> SignedSMsg [format (nir d d d o)] .
op -`(_`) : Msg -> SignedSMsg [format (nib d d d o)] .

sort EmptyList .

op nil : -> EmptyList [ctor format (ni d)] .
op _`,_ : EmptyList EmptyList -> EmptyList
               [ctor assoc id: nil format (d d s d)] .

sort SMsgList .

subsort SMsg  < SMsgList .
subsort EmptyList < SMsgList .
```

```
op _`,_ : SMsgList SMsgList -> SMsgList
               [ctor assoc id: nil format (d d s d)] .

sort SMsgList-L SMsgList-R .

op nil : -> SMsgList-R [ctor] .
op _`,_ : SMsg SMsgList-R -> SMsgList-R
                [ctor format (d d s d) gather (e E)] .
op nil : -> SMsgList-L [ctor] .
op _`,_ : SMsgList-L SMsg -> SMsgList-L
                [ctor format (d d s d) gather (E e)] .

sort FreshSet .

subsort Fresh < FreshSet .

op nil : -> FreshSet [ctor] .
op _`,_ : FreshSet FreshSet -> FreshSet [ctor comm assoc
                                          id: nil] .

sort Strand .

op ::_::`[_|_`] : FreshSet SMsgList-L SMsgList-R -> Strand
              [format (ni d d ni s+++ s--- s+++ d s---)] .

sort StrandSet .

subsort Strand < StrandSet .

op empty : -> StrandSet [ctor] .
op _&_ : StrandSet StrandSet -> StrandSet [ctor assoc comm
                             id: empty format (d d d d)] .

sort Knowledge-!inI Knowledge-inI
     IntruderKnowledge-empty  Knowledge .
```

```
subsort Knowledge-!inI Knowledge-inI
        IntruderKnowledge-empty < Knowledge .

op _!inI : Msg -> Knowledge-!inI [format (ni d o)] .
op _inI : Msg -> Knowledge-inI [format (niu d o)] .

sort IntruderKnowledge-!inI IntruderKnowledge-inI
     IntruderKnowledgeElem  IntruderKnowledge .

subsort IntruderKnowledge-empty < IntruderKnowledge-!inI .
subsort IntruderKnowledge-empty < IntruderKnowledge-inI .
subsort IntruderKnowledge-!inI IntruderKnowledge-inI
        < IntruderKnowledge .
subsort Knowledge-!inI < IntruderKnowledge-!inI .
subsort Knowledge-inI < IntruderKnowledge-inI .
subsort Knowledge < IntruderKnowledgeElem
        < IntruderKnowledge .

op empty : -> IntruderKnowledge-empty [ctor] .
op _`,_ : IntruderKnowledge IntruderKnowledge
          -> IntruderKnowledge
          [ctor assoc comm id: empty format (d d d d)] .
op _`,_ : IntruderKnowledge-!inI IntruderKnowledge-!inI
          -> IntruderKnowledge-!inI [ditto] .
op _`,_ : IntruderKnowledge-inI IntruderKnowledge-inI
          -> IntruderKnowledge-inI [ditto] .
op _`,_ : IntruderKnowledge-empty IntruderKnowledge-empty
          -> IntruderKnowledge-empty [ditto] .

sort System .

op _||_ : StrandSet IntruderKnowledge -> System
          [format (d n d n)] .

sorts Name Nonce Key .

subsort Name Nonce Key < Msg .
```

```
subsort Name < Key .
subsort Name < Public .

--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .

--- Encoding operators for public/private encryption
op pk : Name Msg -> Msg  .
op sk : Name Msg -> Msg  .

--- Associativity operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .

--- constant that denotes the intruder's name (you could
    also have the same for Alice and Bob)
op i : -> Name .
op a : -> Name .
op b : -> Name .
op c0 : -> Fresh .
op c1 : -> Fresh .

var SS : StrandSet .
var K : IntruderKnowledge .
var ML : SMsgList .
var L1 : SMsgList-L .
var L2 : SMsgList-R .
var M   : Msg .
var rrL : FreshSet .
var A B : Name .
vars X Y Z : Msg .
var Sys Sys' : System .
var r r' : Fresh .
var Str : Strand .
var N NA NB : Nonce .
var IK : IntruderKnowledge .
var Ke : Key .
```

```
--- this rule accepts an output message, without
    modifiying the intruder knowledge

rl [acceptOutput] : (SS:StrandSet &
      :: rrL:FreshSet ::
      [ L1:SMsgList-L | +(M:Msg), L2:SMsgList-R])
      || K:IntruderKnowledge
  => (SS:StrandSet &
      :: rrL:FreshSet ::
      [ L1:SMsgList-L, +(M:Msg) | L2:SMsgList-R])
      || K:IntruderKnowledge .

--- this rule accepts an output message that already
    appears in the negative intruder knowledge
--- thus, the term M !inI is modified to M inI to denote
    that the intruder has learnt that message.

rl [acceptOutput2] : (SS:StrandSet &
      :: rrL:FreshSet ::
      [ L1:SMsgList-L | +(M:Msg), L2:SMsgList-R])
      || K:IntruderKnowledge, M:Msg !inI
  => (SS:StrandSet &
      :: rrL:FreshSet ::
      [ L1:SMsgList-L, +(M:Msg) | L2:SMsgList-R])
      || K:IntruderKnowledge, M:Msg inI .

--- this rule accepts an input message and does not
    modify the intruder knowledge

rl [acceptInput] : (SS:StrandSet &
      :: rrL:FreshSet ::
      [ L1:SMsgList-L | -(M:Msg), L2:SMsgList-R])
      || K:IntruderKnowledge
  => (SS:StrandSet &
      :: rrL:FreshSet ::
      [ L1:SMsgList-L, -(M:Msg) | L2:SMsgList-R])
      || K:IntruderKnowledge .
```

```
  --- Encryption/Decryption Cancellation

  eq [eqPK-SK] : pk(Ke,sk(Ke,Z)) = Z .
  eq [eqSK-PK] : sk(Ke,pk(Ke,Z)) = Z .

endm
```

Webmail Application Specified in Web-TLR

The webmail application that we analyze in the Web-TLR example is as follows:

```
mod INITIAL is

  inc STRING + INT + QID .

  op NHISTORY : -> Int .
  op IS-HISTORY : -> Bool .

  eq [NHISTORY] : NHISTORY = 10 .
  eq [IS-HISTORY] : IS-HISTORY = true .

endm



mod TYPES is

  inc INITIAL .

  sorts Value Id .
  subsorts  Int < Value .

  vars v1 v2 : Value .
  vars s1 s2 : String .
  vars n i1 i2 : Int .
```

```
  vars b1 b2 : Bool .
  vars id id1 id2 : Id .

  op _._ : Id Int -> Id [ctor] .
  op newid : Id Int -> Id  [ctor] .

  eq [newid] : newid ( id1  , n) =   id1 . n .

  op null : -> Value [ctor] .
  op i_ : Int -> Value [ctor] .
  op s`(_`) : String -> Value [ctor] .
  op b_ : Bool -> Value [ctor] .
  op _v+_ : Value Value -> Value .

  eq  [a1] : ( s(s1) v+ s(s2) ) = s( s1 + s2 ) .
  eq  [a2] : ( i(i1) v+ i(i2) ) = i( i1 + i2 ) .
  eq  [a3] : v1 v+ v2 = v1 + v2 [owise] .

  op string : Id -> String .

  eq [st] : string (id) = "id" .

endm



mod DATABASE is

  inc TYPES .

  sorts DB SqlDB  ValueDB .

  subsorts Value < SqlDB .
  subsorts Value < ValueDB .

  op sql-empty : -> SqlDB [ctor] .
  op valueDB-empty : -> ValueDB [ctor] .
```

```
  op _:_ : ValueDB ValueDB -> ValueDB [ctor assoc
                                          id: null] .
  op `(_;_`) : SqlDB Value -> DB [ctor] .
  op db-empty : -> DB [ctor] .
  op __ : DB DB -> DB [ctor assoc comm id: db-empty] .

  vars sql : SqlDB .
  vars v v1 v2 : Value .
  vars db dbs : DB .

  op select : DB SqlDB -> ValueDB .

  eq [select1] : select ((sql ; v) dbs , sql )
               = v : (select ( dbs , sql )  ) .
  eq [select2] : select (dbs , sql ) = null [owise] .

  op update : DB SqlDB ValueDB -> DB .

  eq [update1] : update ((sql ; v1) dbs, sql, v2)
               = (sql ; v2) dbs .
  eq [update2] : update ( db, sql, v)
               = (sql ; v) db [owise] .

  op insert : DB SqlDB ValueDB -> DB .

  eq [insert1] : insert ( db, sql, v) = (sql ; v) db .

endm



mod SESSION is

  inc TYPES .

  sorts Session UserSession .
```

```
  vars n : Value .
  vars s ss : Session .
  vars v v1 v2 : Value .

  op `(_`,_`) : Value Value -> Session [ctor] .
  op session-empty : -> Session [ctor] .
  op _:_ : Session Session -> Session
                    [ctor assoc comm id: session-empty] .
  op _in_ : Value Session -> Bool .

  eq [aux1] : n in ((n , v) : ss) = true .
  eq [aux2] : n in  s = false [owise] .

  op getSessionValue : Session Value -> Value .

  eq [getSessionValue1] : getSessionValue(((n , v) : ss), n)
                          = v .
  eq [getSessionValue2] : getSessionValue(s, n)
                          = null [owise] .

  op setSessionValue : Session Value Value -> Session .

  eq [getSessionValue3] : setSessionValue(((n, v1) : ss),
                          n, v2) = ((n, v2) : ss) .
  eq [getSessionValue4] : setSessionValue(s, n, v)
                          = ((n, v) : s) [owise] .

  op us`(_`,_`) : Id Session -> UserSession [ctor]  .
  op usersession-empty : -> UserSession [ctor]  .
  op _:_ : UserSession UserSession -> UserSession
                 [ctor assoc comm id: usersession-empty] .

endm
```

```
mod QUERY is

  inc TYPES .

  sorts Query Sigma Name  .

  vars n : Qid .
  vars v w : String .
  vars q qs : Query .
  vars z zs : Sigma .

  op _'=_ : Qid String -> Query [ctor] .
  op query-empty : -> Query [ctor] .
  op _:_ : Query Query -> Query [ctor assoc comm
                                  id: query-empty] .
  op _in_ : Qid Query -> Bool .

  eq [aux5] : n in ((n '= v) : qs) = true .
  eq [aux6] : n in  q = false [owise] .

  op _/_ : Qid String -> Sigma [ctor] .
  op sigma-empty : -> Sigma [ctor] .
  op _:_ : Sigma Sigma -> Sigma [ctor assoc comm
                                  id: sigma-empty] .
  op sigma : Sigma Query -> Query .

  eq [sigma1] : sigma (((n / v) : zs) , ((n '= w) : qs))
               = ((n '= v) : (sigma(((n / v) : zs), qs))) .
  eq [sigma2] : sigma (zs, qs) =  qs  [owise] .

endm
```

```
mod MEMORY is

  inc TYPES .

  sorts Memory  .

  op none : -> Memory [ctor] .
  op _:_ : Memory Memory -> Memory [ctor assoc comm
                                        id: none] .
  op `[_`,_`] : Qid Value -> Memory [ctor] .
  op _in_ : Qid Memory -> Bool .

  var q : Qid .
  var m : Memory .
  var v : Value .

  eq [axx3] : q in [q,v] : m = true .
  eq [axx5] : q in  m = false [owise] .

endm


mod EXPRESSION is

  inc MEMORY  + QUERY + SESSION + DATABASE .

  sorts Expression Test .

  subsorts Test Value Qid < Expression .

  vars ex ex1 ex2 ex3 ex4 : Expression .
  vars b66 : Bool .
  vars m ms  : Memory .
  vars db dbs : DB .
  vars s ss : Session .
  vars q qs : Query .
  vars x y : Int .
```

```
vars qid : Qid .
vars v  : Value .
vars str : String .
vars sql : SqlDB .
vars t : Test .

op TRUE : -> Test .
op FALSE : -> Test .
op _=_  : Expression Expression -> Test .
op _!=_  : Expression Expression -> Test .
op _'+_ : Expression Expression -> Expression .
op _'*_ : Expression Expression -> Expression .
op _'._ : Expression Expression -> Expression .

--- The expression, memory, session, query, bada base
op eval : Expression Memory Session Query DB
          -> Expression  .

eq [eval1] : eval ( v , m , s , q , db ) = v .
eq [eval2] : eval ( ex , m , s , q , db ) = null [owise] .
ceq [eval3] : eval ( ex1 = ex2 , m , s, q, db ) = TRUE
                 if ex3 := eval(ex1, m, s, q, db)
                    /\ ex4 := eval(ex2, m, s, q, db)
                    /\ ex3 == ex4 .
ceq [eval4] : eval ( ex1 = ex2 , m , s, q, db ) = FALSE
                 if ex3 := eval(ex1, m, s, q, db)
                    /\ ex4 := eval(ex2, m, s, q, db)
                    /\ ex3 =/= ex4 .
ceq [eval5] : eval ( ex1 != ex2 , m , s, q, db ) = FALSE
                 if ex3 := eval(ex1, m, s, q, db)
                    /\ ex4 := eval(ex2, m, s, q, db)
                    /\ ex3 == ex4 .
ceq [eval6] : eval ( ex1 != ex2 , m , s, q, db ) = TRUE
                 if ex3 := eval(ex1, m, s, q, db)
                    /\ ex4 := eval(ex2, m, s, q, db)
                    /\ ex3 =/= ex4 .
eq [eval7] : eval (qid, ([qid, v] : ms), s, q , db) = v .
```

```
  ceq [eval8] : eval ( qid ,  m, s, q, db ) = null
                 if b66 := qid in m
                    /\ b66 =/= true .
  eq [eval9] : eval(ex1 '+ ex2 , m , s, q, db)
             = eval(ex1, m, s, q, db)
                 + eval(ex2, m, s, q, db) .
  eq [eval10] : eval ( ex1 '* ex2 , m , s, q, db )
              = eval(ex1, m, s, q, db)
                 * eval(ex2, m, s, q, db) .
  eq [eval11] : eval ( ex1 '. ex2 , m , s, q, db )
              = eval(ex1, m, s, q, db)
                 + eval(ex2, m, s, q, db) .

  ----- extended for the web -----
  op getSession : Expression -> Expression  .
  op getQuery : Qid -> Expression .

  eq [eval12] :  eval(getSession(ex), m, s, q, db)
               = getSessionValue(s, eval(ex, m,
                                           s, q, db)) .
  eq  [eval13] : eval(getQuery(qid), m, s,
                      (qid '= str) : qs, db) = s(str) .
  ceq [eval14] : eval(getQuery(qid), m, s, q, db) = null
                     if b66 := qid in q
                        /\ b66 =/= true .

  ----- extended for persistente information
  op selectDB : Expression -> Expression .


  eq  [eval15] : eval (selectDB(ex), m, s, q, db)
               = select(db,  eval(ex, m, s, q, db)) .

endm
```

```
mod SCRIPT is

  inc EXPRESSION .

  sorts Script ScriptState .

  op skip : -> Script .
  op _;_ : Script Script -> Script [prec 61 assoc id: skip] .
  op _:=_ : Qid Expression -> Script .
  op if_then_else_fi : Test Script Script -> Script .
  op if_then_fi : Test Script -> Script .
  op while_do_od : Test Script -> Script .
  op repeat_until_od : Script Test -> Script .
  op `[_`,_`,_`,_`,_`] : Script Memory Session Query DB
                       -> ScriptState .

  vars ex ex1 ex2 : Expression .
  vars m ms  : Memory .
  vars b66 : Bool .
  vars db dbs : DB .
  vars s ss : Session .
  vars q qs : Query .
  vars x y : Int .
  vars qid : Qid .
  vars v  : Value .
  vars str : String .
  vars p p1 p2 ps : Script .
  vars t : Test .
  vars sql : SqlDB .

  op eval : ScriptState -> ScriptState .

  --- skip - stop the eval
  eq [eval15] : eval ( [ skip , m , s , q , db ] )
              = [ skip , m , s , q , db ] .
  eq [eval16] : eval ( [ p , m , s , q , db ] )
              = [ p , m , s , q , db ] [owise] .
```

```
--- :=
eq [eval17] : eval ([(qid := ex); ps, [qid, v] : ms, s,
              q, db]) = eval([ps, [qid, eval(ex, [qid, v]
                              : ms, s, q, db)] : ms, s, q,
                              db]) .
ceq [eval18] : eval ( [ (qid := ex ); ps ,  ms, s, q ,
               db ] ) = eval ([ ps , [qid, eval(ex, ms, s,
                              q, db ) ] : ms, s, q, db ])
                   if b66 := qid in ms /\ b66 =/= true .

--- if then else fi
ceq [eval19] : eval ( [ ( if t then p1 else p2 fi ) ; ps,
               m, s, q , db ] ) = eval ([ p1 ; ps , m, s,
                                           q, db ])
                    if TRUE := eval(t, m, s, q, db) .
ceq [eval20] : eval ( [ ( if t then p1 else p2 fi ) ; ps,
               m, s, q , db ] ) = eval ([ p2 ; ps , m, s,
                                             q, db ])
                    if FALSE == eval(t, m, s, q, db) .

--- if then  fi
eq [eval21] : eval ( [ ( if t then p fi ); ps , m, s, q,
              db ] ) = eval ([ ( if t then p
                                     else skip fi )
              ; ps, m, s, q, db ]) .

--- while do od
ceq [eval22] : eval ( [ ( while t do p od ); ps,
               m, s, q , db ] ) = eval ([ p ; while t do
               p od ; ps , m, s, q, db ])
                     if  TRUE := eval(t, m, s, q, db) .

ceq [eval23] : eval ( [ ( while t do p od ); ps , m, s,
               q , db ] ) = eval ([  ps , m, s, q, db ])
                     if  FALSE := eval(t, m, s, q, db) .
```

```
  --- repeat until od
  eq [eval24] : eval ( [ ( repeat p until t od ); ps , m,
                 s, q , db ] ) = eval ([  p ; (while t do
                 p od) ; ps , m, s, q, db ])  .

   --- extended for the web
  op setSession : Expression Expression -> Script .
  op clearSession : -> Script .

  eq [eval25] : eval ([ ( setSession(ex1, ex2) ); ps, m,
                s , q , db ]) = eval ([ ps , m,
                            setSessionValue (s, eval(ex1,
                            m, s, q, db), eval(ex2, m, s,
                            q, db) ), q , db ])  .
  eq [eval26] : eval ([ clearSession ; ps , m, s , q ,
                db ]) = eval ([ ps , m, session-empty , q ,
                            db ])  .

  ----- extended for persistente information
  op updateDB : Expression Expression -> Script .
  eq [eval27] : eval ([(updateDB(ex1, ex2)); ps, m,
                 s, q, db]) = eval ([ ps, m, s, q,
                            update(db, eval(ex1, m, s,
                                    q, db), eval(ex2, m,
                                    s, q, db) ) ])  .

endm


mod CONDITION is

  inc TYPES + SESSION .

  sorts Condition . --- Name Value .

  vars c cs : Condition .
  vars s ss : Session .
```

```
  vars n : Value .
  vars v v1 v2 : Value .

  op TRUE  : -> Condition [ctor] .
  op FALSE : -> Condition [ctor] .
  op cond-empty : -> Condition [ctor] .
  op _'==_ : Value Value -> Condition [ctor] .
  op _:_ : Condition Condition -> Condition
                      [ctor assoc comm id: cond-empty] .
  op holdCondition : Condition Session -> Bool .

  eq [holdCondition1] : holdCondition( cond-empty , s )
                        = true  .
  eq [holdCondition2] : holdCondition( TRUE , s ) = true  .
  eq [holdCondition3] : holdCondition( FALSE , s ) = false  .
  eq [holdCondition4] : holdCondition( (n '== v) : cs,
                                       (n, v) : ss )
                        =  holdCondition(cs, ss) .
  eq [holdCondition5] : holdCondition( c , s )
                        = false [owise]  .

endm


mod WEB_MODEL is

  inc SCRIPT + CONDITION  .

  pr SET{Qid} .

  sorts Page URL Continuation
        Navigation Browser Server
        Message ReadyMessage History .

  vars id idw : Id .
  vars z : Sigma .
  vars np np1 np2 : Qid .
```

```
vars q : Query .
vars sc : Script .
vars wapps : Page .
vars cont : Continuation .
vars nav : Navigation .
vars nat : Nat .
vars h hs : History .
vars url url1 url2 : URL .
vars m m1 m2 : Message .
vars s : Session .

--- URL  -----
--- Page name , query .
op _?_ : Qid Query -> URL [ctor] .
op url-empty : -> URL [ctor] .
op _:_ : URL URL -> URL [ctor assoc comm id: url-empty] .

--- Continuation ---
--- condition , page name .
op `(_=>_`) : Condition Qid -> Continuation [ctor] .
op cont-empty : -> Continuation .
op _:_ : Continuation Continuation -> Continuation
                       [ctor assoc comm id: cont-empty] .

--- Navigation -----
op `(_->_`) : Condition URL -> Navigation [ctor] .
op nav-empty : -> Navigation .
op _:_ : Navigation Navigation -> Navigation
                        [ctor assoc comm id: nav-empty] .

--- WebPage -----
--- Name, script, continuation, navigation .
op `(_`,_`,`{_`}`,`{_`}`) : Qid Script Continuation
                              Navigation -> Page [ctor] .
op page-empty : -> Page .
op _:_ : Page Page -> Page
                       [ctor assoc comm id: page-empty] .
```

```
op pageNotFound : -> Qid .
op notFoundPage : -> Page .

eq notFoundPage = ( pageNotFound, skip,
                    {cont-empty},
                    {nav-empty} ) .

--- History -----
op H : Qid URL Message -> History [ctor] .
op history-empty : -> History [ctor] .
op __ : History History -> History
                    [ctor assoc id: history-empty] .
op _:_ : History History -> History .
op long : History -> Int .

eq [long1] : long (history-empty) = 0 .
eq [long2] : long ( H(np,url,m) : h ) = 1 + long(h) .
ceq [hc] : H(np1,url1,m1) : (hs H(np2,url2,m2))
           = H(np1,url1,m2) hs   if long(hs) = NHISTORY .
eq [h] : h : hs =  h hs  [owise] .

--- Browser -----
--- IdBrowser, IdWindows, NamePage, Urls,
--- Session Database LastMessage History  IdLastMes
op B : Id Id Qid URL Session Sigma Message History Nat
       -> Browser [ctor] .
op br-empty : -> Browser .
op _:_ : Browser Browser -> Browser
                       [ctor assoc comm id: br-empty] .
op PageNameEmpty : -> Qid .

eq [PageNameEmpty] : PageNameEmpty = 'PageNameEmpty .

op URLEmpty : -> URL  .
op brEmpty : Id Id Sigma -> Browser .
```

```
eq [brEmpty] : brEmpty (id, idw, z)
               = B( id, idw, 'PageNameEmpty, url-empty,
                             session-empty, z, mes-empty,
                             history-empty , 1) .

op noPage : -> Qid .

eq [noPage] : noPage = 'noPage .

op newBrowser : Id Id URL Sigma -> Browser .

eq [newBrowser] : newBrowser (id, idw, (np ? q), z)
               = B(id, idw, noPage, (np ? q),
                       session-empty, z, mes-empty,
                       history-empty, 1) .

--- Message -----
op m : Id Id URL Nat -> Message [ctor] .
op m : Id Id Qid URL Session Nat -> Message [ctor] .
op mes-empty : -> Message .
op _:_ : Message Message -> Message
                            [ctor assoc id: mes-empty] .

--- ReadyMessage ---
op rm : Message Session DB -> ReadyMessage [ctor] .
op readymes-empty : -> ReadyMessage .
op _:_ : ReadyMessage ReadyMessage -> ReadyMessage
                     [ctor assoc  id: readymes-empty] .

--- Server -----
op S : Page UserSession Message ReadyMessage DB
       -> Server [ctor] .

--- auxiliary operations
op allWebPages : Page -> Set{Qid} .
```

```
  eq [allWebPages1] : allWebPages (page-empty)
                    = insert(noPage, (insert('PageNameEmpty,
                          empty)))  .
  eq [allWebPages2] : allWebPages (( np , sc , { cont } ,
                                     { nav } ) : wapps )
                     = insert( np, allWebPages ( wapps ) ) .
endm


mod EVAL is
  inc WEB_MODEL  .

  vars page wapp wapps w : Page .
  vars np qid np1 np2 nextPage : Qid .
  vars q q1 : Query .
  vars sc sc1 : Script .
  vars cont conts : Continuation .
  vars nav : Navigation .
  vars ss nextS : Session .
  vars cond conds : Condition .
  vars url urls nextURLs : URL .
  vars id idw : Id .
  vars uss : UserSession .
  vars db nextDB : DB .
  vars m : Memory .
  vars idmes : Nat .

  op pageNotContinuaton : -> Qid .

  --- eval the continuation
  op  holdContinuation : Qid Continuation Session -> Qid .

  eq  [holdContinuation1] : holdContinuation(np, cont-empty,
                                ss) = np .
```

```
eq  [holdContinuation2] : holdContinuation(np,
                                (cond => np) : conts, ss)
                          = holdCont (np, (cond => np)
                               : conts , ss) .
ceq [holdContinuation3] : holdContinuation(np, conts, ss)
                          = qid
               if np1 :=  holdCont (np, conts , ss)
                  /\ qid := whichQid(np, np1) [owise] .

op  holdCont : Qid Continuation Session -> Qid .

eq  [holdCont1] : holdCont (np, cont-empty, ss)
                 = pageNotContinuaton .
ceq [holdCont2] : holdCont (np, (cond => qid) : conts, ss)
                 = qid  if ( holdCondition(cond,ss) ) .
eq  [holdCont3] : holdCont (np, (cond => qid) : conts, ss)
                 = holdCont (np, conts, ss) [owise] .

op whichQid : Qid Qid -> Qid .

eq [whichQid1] : whichQid ( np , pageNotContinuaton )
                 = np .
eq [whichQid2] : whichQid ( np , np1 ) = np1 [owise] .

--- eval the navegation
op holdNavigation : Qid Page Session -> URL .

eq [holdNavigation1] : holdNavigation(np, (( np, sc,
              { cont }, { nav } ) : wapp ), ss  )
              = getURLs (nav, ss) .
eq [holdNavigation2] : holdNavigation(np ,  wapp , ss  )
              = url-empty [owise] .

op  getURLs : Navigation Session -> URL .

eq  [getURLs1] : getURLs ( nav-empty , ss ) = url-empty .
```

```
  ceq [getURLs2] : getURLs ( ( cond -> url) : nav , ss )
                   = url : getURLs ( nav , ss )
                if ( holdCondition(cond,ss) ) .
  eq  [getURLs3] : getURLs ( ( cond -> url) : nav , ss )
                   = getURLs ( nav , ss )  [owise] .

  --- eval the script
  op  evalScript : Page UserSession Message DB
      -> ReadyMessage .

  ceq [evalScript1] : evalScript (
         (( np , sc , { cont } , { nav } ) : wapps ) ,
         us( id, ss ) : uss ,
         m( id , idw , (np ? q) , idmes ) ,
         db )
      = rm( m( id, idw, nextPage, nextURLs, nextS, idmes),
             nextS, nextDB)
      if [sc1, m, nextS, q1, nextDB] := eval([sc, none, ss,
         q, db] )
         /\ nextPage := holdContinuation (np, cont, nextS)
         /\ nextURLs := holdNavigation (nextPage, (( np ,
         sc , { cont } , { nav } ) : wapps ) , nextS) .

  eq [evalScript2] : evalScript ( wapp , us( id, ss ) :
         uss , m( id, idw, (np ? q), idmes ) , db  )
      = rm( m( id, idw, pageNotFound, url-empty, ss,
         idmes ), ss, db ) [owise] .

endm
```

```
mod BROWSER-ACTION is

  inc INT + QID .

  sorts BrowserActions Tab .

  vars np : Qid .
  vars ba : BrowserActions .
  vars ln tab f5 : Qid .

  op T : Int Qid -> Tab [ctor] .
  op tab-empty : -> Tab .
  op _;_ : Tab Tab -> Tab [ctor assoc comm id: tab-empty] .
  op f5-empty : -> Qid .
  op _;_ : Qid Qid -> Qid [ctor assoc comm id: f5-empty] .
  op bra-empty :  -> BrowserActions .
  op BA : Tab Qid -> BrowserActions [ctor] .

endm


mod PROTOCOL is

  inc WEB_MODEL + EVAL + BROWSER-ACTION .

  sorts WebState .

  op `[_`]_`[_`]`[_`] : Browser BrowserActions Message
           Server -> WebState [ctor] .

  vars id idw : Id .
  vars p wapp : Page .
  vars np np1 np2 f5 : Qid .
  vars q q1 : Query .
  vars z : Sigma .
  vars urls urls1 urls2 : URL .
  vars ms ms1 pms lms : Message .
```

```
vars rms : ReadyMessage .
vars sv : Server .
vars ss ss1 ss2 : Session .
vars uss : UserSession .
vars brs : Browser .
vars db : DB .
vars h : History .
vars idlm idlm1 idlm2 : Nat .
vars ba : BrowserActions .
vars n : Int .
vars tab : Tab .

op evalScriptRM : -> ReadyMessage .
op block-db : -> DB [ctor] .
op createUserSession : Id Server -> Server .

eq [createUserSession1] : createUserSession (id , S(wapp ,
                us( id, ss ) : uss  ,  pms , rms, db)  ) =
                S(wapp , us( id, ss ) : uss  ,  pms ,
                   rms, db ) .
eq [createUserSession2] : createUserSession (id , S(wapp ,
                uss  ,  pms , rms, db)  ) =
                S(wapp , us( id, session-empty )
                   : uss  ,  pms , rms, db ) [owise] .

--- definition of the protocol request-response -----
--- request.click
--- browser submit a request to the server.
rl [createSession] : [ B(id, idw, np, (np1 ? q1) : urls,
                        ss, z, lms, h, idlm) : brs ] ba
       [ ms ]
       [ S(wapp , usersession-empty ,  pms , rms, db ) ]
 =>
       [ B(id, idw, np, (np1 ? q1) : urls, ss, z, lms,
           h, idlm) : brs ] ba
       [ ms ]
       [ S(wapp , us( id, session-empty ) ,  pms , rms,
```

```
             db ) ] .

rl [ReqIni] : [ B(id, idw, np, (np1 ? q1) : urls, ss, z,
                  lms, h, idlm) : brs ] ba
              [ ms ] [ sv ]
        =>
           [ B( id, idw, 'PageNameEmpty, url-empty , ss, z,
               m(id, idw, (np1 ? (sigma(z,q1))), idlm ), h,
                  idlm)
                   : brs ] ba
           [ ms : m(id, idw, (np1 ? (sigma(z,q1))), idlm) ]
           [ sv ] .

--- request.read
--- the server read the message and add to the set
--- of pending messages.
rl [ReqFin] :
           [ brs ] ba
     [ m(id, idw, urls, idlm) : ms ]
          [ S(wapp , uss , pms, rms, db) ]
     =>
          [ brs ] ba
          [ ms ]
          [ S(wapp, uss, pms : m(id, idw, urls, idlm),
                rms, db) ] .

--- eval script
--- read a pending message and running the script, and
--- then add to set of ready messages.
rl [ScriptEval] :
          [ brs ] ba
          [ ms ]
          [ S(wapp , uss , m(id, idw, urls, idlm) : pms,
            rms, db) ]
     =>
          [ brs ] ba
          [ ms ]
```

```
          [ S(wapp , uss , pms , rms : evalScript (wapp,
                  us( id, session-empty ), m(id, idw, urls,
                        idlm) , db ) , block-db ) ] .

--- response
--- the server update the session of browser and db,
--- then  sent a (pending) message to the browser.
rl [ResIni] :
        [ brs ] ba
        [ ms ]
        [ S(wapp , us( id, ss1 ) : uss, pms,
             rm(m( id, idw, np , urls, ss2,
                  idlm), ss2, db)  : rms , block-db ) ]
    =>
        [ brs ] ba
        [ ms :  m( id, idw, np , urls, ss2, idlm) ]
        [ S(wapp, us(id, ss2) : uss, pms, rms, db ) ] .

--- response
--- the browser read a message for selft and show
--- the page.
rl [ResFin] :
        [ B(id, idw, np1, urls1, ss, z, lms, h,
             idlm1 ) : brs ] ba
        [ m(id, idw, np2, urls2, ss2, idlm1) : ms ]
        [ sv ]
 =>
        [ B(id, idw, np2, urls2, ss2, z, lms, h,
             idlm1) : brs ] ba
        [ ms ]
        [ sv ] .

--- is not the last message, make a top
rl [ResFinNo] :
        [ B(id, idw, np1, urls1, ss, z, lms, h,
             idlm1) : brs ] ba
        [ m(id, idw, np2, urls2, ss2, idlm2) : ms ]
```

```
          [ sv ]
    =>
          [ B(id, idw, np1, urls1, ss, z, lms, h,
               idlm1) : brs ] ba
          [ ms ]
          [ sv ] .

  --- operations of browser -----
  --- new tab
  rl [newT] :
          [ B(id, idw, np,  urls, ss, z, lms, h,
               idlm) : brs ]  BA( ( T(n,np) ; tab), f5 )
          [ ms ]
          [ sv ]
      =>
          [ B(id, idw, np,  urls, ss, z, lms, h, idlm ) :
               B(id, newid(idw,n), np,  urls, ss, z,
                  mes-empty, history-empty, idlm )  :
                  brs ]  BA( tab , f5 )
          [ ms ]
          [ sv ] .

  --- refresh F5
  rl [F5] :
          [ B(id, idw, np,  urls, ss, z, m(id, idw,
               urls, idlm), h, idlm) : brs ] BA(
                  tab, (np ; f5) )
          [ ms ]
          [ sv ]
      =>
          [ B(id, idw, np,  urls, ss, z, m(id, idw,
               urls, (idlm + 1) ), h, (idlm + 1) )
                   : brs ] BA( tab, f5 )
          [ ms : m(id, idw, urls, (idlm + 1) ) ]
          [ sv ] .

endm
```

```
mod WEBAPP is

  inc PROTOCOL .

  --- Webmail Application
  ops WELCOME HOME EMAIL-LIST
      VIEW-EMAIL CHANGE-ACCOUNT
      ADMINISTRATION ADMIN-LOGOUT
      LOGOUT : -> Qid .

  eq [WELCOME] : WELCOME = 'Welcome .
  eq [HOME] : HOME = 'Home .
  eq [EMAIL-LIST] : EMAIL-LIST = 'Email-list .
  eq [VIEW-EMAIL] : VIEW-EMAIL = 'View-email .
  eq [CHANGE-ACCOUNT] : CHANGE-ACCOUNT = 'Change-account .
  eq [ADMINISTRATION] : ADMINISTRATION = 'Administration .
  eq [ADMIN-LOGOUT] : ADMIN-LOGOUT = 'Admin-Logout .
  eq [LOGOUT] : LOGOUT = 'Logout .

  --- Welcome page
  op welcomePage : -> Page .

  eq [welcomePage] : welcomePage = ( WELCOME , skip ,
                  { cont-empty } ,
                  { ( TRUE  -> ( HOME ? ('user '= "") :
                      ('pass '= "")  ) ) } ) .

  --- Home page
  op homePage : -> Page .

  eq [homePage] : homePage
                = ( HOME , sHome ,
                  {(( s("login") '== s("no") )
                     => WELCOME ) :
                    (( s("changeLogin") '== s("no") )
```

```
                         => CHANGE-ACCOUNT ) :
                        ((s("login") '== s("ok")) => HOME)} ,
                      {(TRUE -> CHANGE-ACCOUNT ? query-empty)
                       : (( s("role") '== s("admin") )
                       -> (ADMINISTRATION ? query-empty)) :
                       (TRUE -> EMAIL-LIST ? query-empty) :
                       (TRUE -> LOGOUT ? query-empty)}) .

op sHome : -> Script .

eq [sHome] : sHome =
      'login := getSession( s("login") ) ;
   if ( 'login = null ) then
        'u := getQuery('user) ;
      'p := getQuery('pass) ;
      'p1 := selectDB('u) ;
      if ( 'p = 'p1 ) then
         'r := selectDB('u '. s("-role") ) ;
         setSession( s("user") , 'u ) ;
     setSession( s("role") , 'r ) ;
     setSession( s("login") , s("ok") )
      else
         setSession( s("login") , s("no") )
      fi
    fi .

--- EmailList page
op emailListPage : -> Page .

eq [emailListPage] : emailListPage
                   = ( EMAIL-LIST, sEmailList ,
                    { cont-empty } ,
                    { ( TRUE
                    -> (VIEW-EMAIL ? ('idEmail '= ""))) :
                    ( TRUE
                    -> ( HOME ? query-empty))}) .
```

```
op sEmailList : -> Script .

eq [sEmailList] : sEmailList =
       'u  := getSession(s("user")) ;
 'es := selectDB( 'u '. s("-email") ) ;
 setSession( s("idEmails-found") , 'es ) .

--- View email page
op viewEmailPage : -> Page .

eq [viewEmailPage] : viewEmailPage = ( VIEW-EMAIL,
                 sViewEmail, { cont-empty } ,
                 { (TRUE -> (EMAIL-LIST ? query-empty)) :
                 ( TRUE -> (HOME ? query-empty))}) .

op sViewEmail : -> Script .

eq [sViewEmail] : sViewEmail =
       'u := getSession(s("user")) ;
 'id := getQuery('idEmail) ;
 'e := selectDB('id ) ;
 setSession( s("text-email") , 'e ) .

--- Change account page
op changeAccountPage : -> Page .

eq [changeAccountPage] : changeAccountPage
                      = ( CHANGE-ACCOUNT , skip ,
                      { cont-empty } ,
                      { ( TRUE  -> ( HOME ?
                      ('newUser '= "") :
                      ('newPass '= "")))}) .

--- Administration page
op administrationPage : -> Page .

eq [administrationPage] : administrationPage
```

```
                   = ( ADMINISTRATION , sAdministration ,
                      { (( s("adminPage") '== s("busy"))
                        => HOME ) } ,
                      { ( TRUE  ->
                        (ADMIN-LOGOUT ? query-empty))}) .

op sAdministration : -> Script .

eq [sAdministration] : sAdministration =
       'adminPage := selectDB( s("adminPage") ) ;
 if ('adminPage = s("free") )  then
            updateDB(s("adminPage") ,
            getSession(s("user")) );
      setSession(s("adminPage") , s("free") )
 else
      setSession(s("adminPage") , s("busy") )
 fi .

--- Admin logout page
op adminLogoutPage : -> Page .

eq [adminLogoutPage] : adminLogoutPage
                   = ( ADMIN-LOGOUT , sAdminLogout ,
                      {(TRUE => HOME)}, {nav-empty}) .

op sAdminLogout : -> Script .

eq [sAdminLogout] : sAdminLogout =
       updateDB(s("adminPage") , s("free") ) .

--- Logout page
op logoutPage : -> Page .

eq [logoutPage] : logoutPage = ( LOGOUT, sLogout,
                   {(TRUE => WELCOME)}, {nav-empty}) .

op sLogout : -> Script .
```

```
  eq [sLogout] : sLogout = clearSession .

  ---- Web Application
  op wapp : -> Page .

  eq [wapp] : wapp =  welcomePage : homePage :
                        emailListPage : viewEmailPage :
                        changeAccountPage :
                        administrationPage :
                        adminLogoutPage : logoutPage .

endm
```

The Pathway Logic Example

The Pathway logic example that models responses to signal stimulation in epithelial-like cells is specified in Maude as follows:

```
fmod PROTEIN is

  pr NAT .

  inc META-LEVEL .

  sorts AminoAcid Protein .

  subsort AminoAcid < Protein .

  ops T Y S K P N : -> AminoAcid .
  ops pT pY pS : -> AminoAcid .

endfm
```

```
fmod THING is

  pr PROTEIN .

  inc META-LEVEL .

  sort Thing Family Composite
       DNA Complex Chemical
       Signature Stimulus .

  subsorts Protein Family Composite
           DNA Complex Chemical
           Signature Stimulus < Thing .

  op (_:_) : Thing Thing -> Complex [assoc comm] .
  op reduce : Complex -> Complex .

  eq [reduce546] : reduce((T1:Thing : T2:Thing))
                   = (T1:Thing : T2:Thing) .

endfm


fmod SOUP is

  pr THING .

  inc META-LEVEL .

  sort Soup .

  subsort Thing < Soup .

  op empty : -> Soup .
  op __ : Soup Soup -> Soup [assoc comm id: empty ] .
  op _has_ : Soup Thing -> Bool .
```

```
  ceq [r2152a] : (T1:Thing S:Soup ) has T2:Thing
         = true if T1:Thing == T2:Thing .
  ceq [r2152b] : (T1:Thing S:Soup ) has T2:Thing
         = S:Soup has T2:Thing if T1:Thing =/= T2:Thing .
  eq [r2654] : (S:Soup has T:Thing) =  false [owise] .

  op # : Nat Thing -> Soup .

  eq [r3456] : #((s N:Nat),T:Thing)
         = T:Thing #(N:Nat,T:Thing) .
  eq [r3256] : #(0,T:Thing) = empty .

  op <_> : Soup -> Complex .
  op reduceS : Soup -> Soup .

  eq [reduceS1] : reduceS(T:Thing T:Thing S:Soup)
         =  reduceS(T:Thing S:Soup) .
  eq [reduceS2] : reduceS(S:Soup) = S:Soup [owise] .
  eq [reduce] : reduce(< s:Soup >) = < reduceS(s:Soup) > .

endfm


fmod MODIFICATION is

  pr SOUP .

  inc META-LEVEL .

  sorts Site Modification ModSet .

  subsort Modification < ModSet .

  op acetyl : -> Modification .
  op acetyl : Site -> Modification .
  op act : -> Modification .
```

```
op act1 : -> Modification .
op act2 : -> Modification .
op act3 : -> Modification .
op bound :      -> Modification .
op bound : Site -> Modification .
op deact : -> Modification .
op degraded : -> Modification .
op dim      : -> Modification .
op disrupted : -> Modification .
op downreg : -> Modification .
op hydrox : -> Modification .
op hydrox : Site -> Modification .
op inhib : -> Modification .
op mem  : -> Modification .
op mito : -> Modification .
op nm : Site    -> Modification .
op notthere : -> Modification .
op out : -> Modification .
op phos :       -> Modification .
op phos : Site  -> Modification .
op phosbound : Site -> Modification .
op pro : -> Modification .
op reloc : -> Modification .
op Sphos : -> Modification .
op STphos : -> Modification .
op STphos1 : -> Modification .
op STphos2 : -> Modification .
op Tphos : -> Modification .
op trunc : -> Modification .
op ubiq : -> Modification .
op Yphos : -> Modification .
ops GTP GDP : -> Modification .
ops GTP GDP : Site -> Modification .
ops mono poly : -> Modification .
ops on off : -> Modification .
ops open closed : -> Modification .
ops ox red    : -> Modification .
```

```
  ops new : -> Modification .
  op num : Nat -> Modification .
  op none : -> ModSet .
  op __ : ModSet ModSet -> ModSet [assoc comm id: none] .
  op __ : AminoAcid Nat -> Site .
  op _contains_ : ModSet Modification -> Bool .

  var M M' : Modification .
  var MS : ModSet .

  eq [e2384] : none contains M' = false .
  ceq [e9573a] : (M MS) contains M' = true if M == M' .
  ceq [e9573b] : (M MS) contains M'
           = MS contains M' if M =/= M' .

  op [_-_] : Protein   ModSet -> Protein   .
  op [_-_] : Family    ModSet -> Family   .
  op [_-_] : Composite ModSet -> Composite   .
  op [_-_] : DNA       ModSet -> DNA       .
  op [_-_] : Chemical  ModSet -> Chemical  .

  var Ptin : Protein .
  var Fmly : Family .
  var Cmsite : Composite .
  var DnaVar : DNA .
  var Cmcal : Chemical .

  eq [e2322n] : [Ptin - none] = Ptin .
  eq [e2556v] : [Fmly - none] = Fmly .
  eq [e2557e] : [Cmsite - none] = Cmsite .
  eq [e2792d] : [DnaVar - none] = DnaVar .
  eq [e2422c] : [Cmcal - none] = Cmcal .

endfm
```

```
fmod LOCATION is

  inc MODIFICATION .

  inc META-LEVEL .

  sort Location LocName .

  subsort Location < Soup .

  op {_|_} : LocName Soup -> Location
                              [format (n d d t d d)] .
  ops CLo CLm CLi CLc : -> LocName .
  ops NUo NUm NUi NUc : -> LocName .
  ops MOo MOm MOi MOc : -> LocName .
  ops MIo MIm MIi MIc : -> LocName .
  ops ERo ERm ERi ERc : -> LocName .
  ops PXo PXm PXi PXc : -> LocName .
  ops GAo GAm GAi GAc : -> LocName .
  ops LEo LEm LEi LEc : -> LocName .
  ops EEo EEm EEi EEc : -> LocName .
  ops LYo LYm LYi LYc : -> LocName .
  ops CPo CPm CPi CPc : -> LocName .
  op  PTc             : -> LocName .
  op  Sig             : -> LocName .

endfm


fmod CELL is

  inc LOCATION .

  inc META-LEVEL .

  sort Cell CellType .
```

```
  subsort Cell < Soup .

  op [_|_] : CellType Soup -> Cell .
  op Cell : -> CellType .
  op EpithelialCell : -> CellType .
  op EverythingCell : -> CellType .
  op Fibroblast : -> CellType .
  op HMEC : -> CellType .
  op LiverCell : -> CellType .
  op Macrophage : -> CellType .
  op MuscleCell : -> CellType .

endfm


fmod DISH is

  inc CELL .

  inc META-LEVEL .

  sort Dish .

  op PD : Soup -> Dish .

endfm



fmod THEOPS is

  inc DISH .

  inc META-LEVEL .

endfm
```

```
fmod PROTEINOPS is

  inc DISH .

  inc META-LEVEL .

  sort ErbB1L .

  subsort ErbB1L < Protein .

  op 1433x1 : -> Protein .
  op Cbl : -> Protein .
  op DAG : -> Chemical .
  op Egf : -> ErbB1L .
  op EgfR : -> Protein .
  op ErbB2 : -> Protein .
  op Gab1   : -> Protein .
  op Grb2 : -> Protein .
  op Hras : -> Protein .
  op IP3 : -> Chemical .
  op Pak1 : -> Protein .
  op Pi3k : -> Composite .
  op PIP2 : -> Chemical .
  op PIP3 : -> Chemical .

  sort Plc .

  subsort Plc < Protein .

  op Plcg : -> Plc .
  op Plcg1 : -> Protein .
  op Plcg2 : -> Protein .
  op PP2a : -> Composite .
  op Raf1 : -> Protein .
  op Shc : -> Protein .
  op Sos1  : -> Protein .
  op Src : -> Protein .
```

```
  op Tgfa   : -> ErbB1L .
  op Ube2l3 : -> Protein .
  op Vav2 : -> Protein .

endfm


mod ALLBP is

  inc PROTEINOPS .

  inc META-LEVEL .

  var  cell : CellType .
  vars clo clm cli clc nuo num nui nuc : Soup .
  vars moo mom moi moc mio mim mii mic : Soup .
  vars ero erm eri erc pxo pxm pxi pxc : Soup .
  vars gao gam gai gac lyo lym lyi lyc : Soup .
  vars eeo eem eei eec leo lem lei lec : Soup .
  vars cpo cpm cpi cpc ct ptc sig : Soup .
  var ms : ModSet .

  rl [1.EgfR.act] :
    ?ErbB1L:ErbB1L
    [CellType:CellType | ct
    {CLm | clm EgfR                     } ]
    =>
    [CellType:CellType | ct
    {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)} ] .

  rl [2.EgfR.ubiq] :
    {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
    {CLi | cli [Cbl - Yphos] [Ube2l3 - ubiq] }
    =>
    {CLm | clm ([EgfR - ubiq] : ?ErbB1L:ErbB1L) }
    {CLi | cli [Cbl - Yphos]  Ube2l3         } .
```

```
rl [4.Gab1.Yphosed] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli [Grb2 - reloc]                 }
  {CLc | clc Gab1                           }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli [Grb2 - reloc] [Gab1 - Yphos] }
  {CLc | clc                                } .

rl [5.Grb2.reloc] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)  }
  {CLi | cli                   }
  {CLc | clc Grb2              }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli [Grb2 - reloc]    }
  {CLc | clc                   } .

rl [6.Hras.act.1] :
  {CLm | clm PIP3                                          }
  {CLi | cli [Grb2 - reloc] [Sos1 - reloc] [Hras - GDP]}
  =>
  {CLm | clm PIP3                                          }
  {CLi | cli [Grb2 - reloc] [Sos1 - reloc] [Hras - GTP]} .

rl [7.IP3.from.PIP2.by.Plc] :
  {CLm | clm PIP2            }
  {CLi | cli [?Plc:Plc - act] }
  {CLc | clc                 }
  =>
  {CLm | clm DAG             }
  {CLi | cli [?Plc:Plc - act] }
  {CLc | clc IP3             } .

rl [8.Pi3k.act] :
  {CLi | cli [Gab1 - Yphos]             }
  {CLc | clc Pi3k                       }
```

```
  =>
  {CLi | cli [Gab1 - Yphos] [Pi3k - act] }
  {CLc | clc                             } .

rl [9.PIP3.from.PIP2.by.Pi3k] :
  {CLm | clm  PIP2        }
  {CLi | cli [Pi3k - act] }
  =>
  {CLm | clm  PIP3        }
  {CLi | cli [Pi3k - act] } .

rl [10.Plcg.act] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) PIP3 }
  {CLi | cli Src             }
  {CLc | clc Plcg            }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) PIP3 }
  {CLi | cli Src [Plcg - act]  }
  {CLc | clc                 } .

rl [11.Shc.Yphosed] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli Src             }
  {CLc | clc Shc             }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli Src [Shc - Yphos] }
  {CLc | clc                 } .

rl [12.Sos1.reinit] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)  }
  {CLi | cli [Grb2 - reloc] [Sos1 - reloc]  }
  {CLc | clc                                }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)  }
  {CLi | cli [Grb2 - Yphos]          }
  {CLc | clc  Sos1                          } .
```

```
rl [13.Sos1.reloc] :
  {CLi | cli [Grb2 - reloc]                }
  {CLc | clc Sos1                          }
  =>
  {CLi | cli [Grb2 - reloc] [Sos1 - reloc] }
  {CLc | clc                               } .

rl [15.Cbl.reloc.Yphos] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli               }
  {CLc | clc Cbl           }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)  }
  {CLi | cli [Cbl - Yphos] }
  {CLc | clc               } .

rl [E56.Pak1.irt.Egf] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli                                }
  {CLc | clc   Pak1                         }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli  [Pak1 - act]                  }
  {CLc | clc                                } .

rl [280.Raf1.by.Hras] :
  {CLi | cli [Hras - GTP] [Pak1 - act] Src }
  {CLc | clc  Raf1           1433x1 PP2a         }
  =>
  {CLi | cli [Hras - GTP] [Pak1 - act]
             Src [Raf1 - act] 1433x1  }
  {CLc | clc                        PP2a         } .

rl [14.Vav2.act] :
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)  }
```

```
    {CLi | cli Src              }
    {CLc | clc Vav2             }
    =>
    {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
    {CLi | cli Src [Vav2 - act] }
    {CLc | clc                  } .

  rl [clt1.ctest] :
    {CLi | cli Src  [Vav2 - act]   [Cbl - Yphos]}
    =>
    {CLi | cli ( Src : ( [Vav2 - act] : [Cbl - Yphos]))} .

endm
```