UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

A Conditional Slicing Tool for Maude

Final Year Project

Ingeniería Informática

Author:
Julia Sapiña Sanchis

Supervisors:
María Alpuente Frasnedo
Francisco Frechina Navarro

Valencia, September 2012

Abstract

In this work we develop JULIENNE, an online trace slicer for the high performance rewriting logic language Maude.

Trace slicing is a widely used technique for execution trace analysis that is effectively used in program debugging, analysis and comprehension. Our slicing tool allows us to systematically trace back reverse dependences and causality along Maude computation traces by means of an algorithm that dynamically simplifies the trace and elides useless data that do not influence the final result.

We describe the main facilities provided by the tool as well as the features and architecture of the tool. The tool is particularly suitable for analyzing complex, textually large execution traces such as those delivered by Maude model checkers.

Keywords

Slicing, Rewriting Logic, Maude, Julienne.

Contents

Introduction **9**

1 Slicing **11**
- 1.1 What is Slicing? 11
- 1.2 Types of Slicing 12
  - 1.2.1 Backward Trace Slicing 13
  - 1.2.2 Incremental Slicing 14

2 JULIENNE Online Trace Slicer **15**
- 2.1 Providing the Maude Specification 15
- 2.2 Providing the Execution Trace 15
- 2.3 Navigating through the Trace 16
- 2.4 Specifying the Slicing Criterion 17
- 2.5 Performing the Slice 18
- 2.6 Trace Slice Results 19

3 Implementation **22**
- 3.1 Technologies 22
  - 3.1.1 Maude 22
  - 3.1.2 HyperText Mark-up Language 23
  - 3.1.3 Cascading Style Sheets 23
  - 3.1.4 JavaScript 24
- 3.2 Implementation Details 25
  - 3.2.1 Graphical User Interface 25
    - 3.2.1.1 General Decisions 25
    - 3.2.1.2 Slider 26
    - 3.2.1.3 Hiding Irrelevant Data 26
  - 3.2.2 Selecting the Slicing Criterion 27
    - 3.2.2.1 Subterm position 28
    - 3.2.2.2 Common technical details 29
    - 3.2.2.3 Meta-level representation 30
    - 3.2.2.4 Source-level representation 31

3.2.2.5 State Map Structure 32

4 Using JULIENNE **35**
4.1 Crossing River Example 35
4.2 Bank Example 40

Conclusions and Future Work **47**

Bibliography **48**

List of Figures

2.1 JULIENNE showing a list of predefined examples and loading the *Bank with error example* specification. 16
2.2 Maude execution trace loaded in JULIENNE. 17
2.3 Generation of a Maude execution trace by providing both the initial and final state. 18
2.4 JULIENNE flashing in red and alerting the user about a possible mistake in the provided information. 19
2.5 Information displayed (in source-level representation) after pressing state transition arrow. 20
2.6 Slicing criteria fixed in the states 20 and 21 of the execution trace. 20
2.7 Result obtained after applying the backward-slicing technique with the slicing criterion described in Figure 2.6. 21

3.1 Original versus sliced trace. 27
3.2 Position tree of a state expressed in meta-level representation. 29
3.3 Position tree of a state expressed in source-level representation. 30

4.1 JULIENNE showing a specification of the crossing river example. 36
4.2 Generation of a Maude execution trace by providing the initial and final state in source-level representation. 37
4.3 A Maude execution trace of 56 states with a selected slicing criterion in the last state. 37
4.4 Original and sliced trace states after applying the backward-slicing technique with the slicing criterion described in Figure 4.3. 38
4.5 Sliced trace showing a reduction of 77% with respect to the original execution trace. 39
4.6 JULIENNE showing a Maude bank specification with an error in one of its rules. 40
4.7 Execution trace provided as an example for the faulty Bank specification. 41

4.8 A 22 state execution trace of the faulty Bank specification with the minus symbol selected as the slicing criterion. 42
4.9 Snapshot of the tool after applying the backward-slicing technique in Figure 4.8. 43
4.10 Original versus sliced trace for the Bank example. 44
4.11 States 8 and 9 of the execution trace displayed in Figure 4.9 with a slicing criterion specified in state 9. 45
4.12 Snapshot of JULIENNE after applying the backward-slicing technique in Figure 4.11. 45
4.13 Table of results of Figure 4.12. 46

List of Tables

3.1 Breakdown of a map by elements and their correspondence with the associated state. 33

Introduction

Software systems commonly generate large and complex execution traces, whose analysis (or even simple inspection) is extremely time-consuming and, in some cases, is not feasible to perform by hand. Trace slicing is a technique that simplifies execution traces by focusing on selected execution aspects, which makes it well suited to program analysis, debugging, and monitoring [7].

Rewriting Logic (RWL) is a very general *logical* and *semantic framework* that is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [6] and Web systems [2, 5]). RWL is efficiently implemented in the high-performance system Maude [8]. Rewriting logic-based tools, like the Maude-NPA protocol analyzer, Maude LTLR model checker, and the Java PathExplorer runtime verification tool (just to mention a few [13]), are used in the analysis and verification of programs and protocols wherein the states are represented as algebraic entities that use equational logic and the transitions are represented using conditional rewrite rules. These transitions are performed *modulo* conditional equational theories that may also contain algebraic axioms such as commutativity and associativity. The execution traces produced by such 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 for analyzing a given piece of information in a given state of the trace. For instance, consider the following rules that define (a part of) the standard semantics of a simple imperative language: 1) `crl <while B do I, St> => <skip, St> if <B, St> => false /\ isCommand(I)`, 2) Then, in the 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 replacing `X := X + 1` with a special variable • and by enforcing the compatibility condition `isCommand(•)`. This condition guarantees the correctness of the simplified trace [4]. In other words, any concretization of the simplified trace (which instantiates the variable • and meets the compatibility condition) is a valid trace that still generates the target data that we are observing (in this case, the output {}).

The JULIENNE slicing tool [1] is based on the conditional slicing technique described in [4] that slices an input execution trace with regard to a set of *target symbols* (which occur in a selected state of the trace), by propagating them backwards through the trace so that all pieces of information that are not an antecedent of the target symbols are simply discarded. Unlike standard backward tracing approaches, which are based on a costly, dynamic labelling procedure [3, 12], in [4], the relevant data are traced back by means of a less expensive, incremental technique of matching refinement. JULIENNE generalizes and supersedes a previous unconditional slicer mentioned in [3]. The system copes with the extremely rich variety of % program conditions that 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 those rewrite steps are revisited backwards in an instrumented, fine-grained way.

In order to formally guarantee the strong correctness of the generated trace slice, the instantiated conditions of the equations and rules are recursively processed, which may imply slicing a number of (originally internal) execution traces, and a Boolean compatibility condition is carried, which ensures the executability of the sliced rewrite steps.

Chapter 1

Slicing

1.1 What is Slicing?

Slicing is a term introduced by Mark Weiser in 1979 to describe a technique that elides irrelevant segments of a program with respect to a point of interest, referred to as the slicing criterion.

The main goal of this technique is to isolate all the segments of a program that contribute to determine the state of the specified slicing criterion into a much more lightened new program, while preserving all the possible control and data dependences. Furthermore, both original and sliced programs must behave exactly the same as the slicing criterion is concerned.

The following is a very simple example that illustrates this concept. Given the following program excerpt:

```
(1) a = 1
(2) b = 2
(3) c = 3
(4) d = a + c
(5) f = b + c
(6) g = d + f
```

The result of slicing it by using `d` as the slicing criterion is:

```
(1) a = 1
(2) b = 2
(3) c = 3
(4) d = a + c
(5) f = b + c
(6) g = d + f
```

as `b`, `f` and `g` do not affect the value of `d`. By using instead `f` as the slicing criterion the result would be:

```
(1) a = 1
(2) b = 2
(3) c = 3
(4) d = a + c
(5) f = b + c
(6) g = d + f
```

As before, we can remove the lines of code that do not affect the slicing criterion, which in this case are the first, fourth and sixth. Finally, by using `g` as the slicing criterion the result witnesses the statement that all programs are slices of themselves, albeit possibly not the most useful one:

```
(1) a = 1
(2) b = 2
(3) c = 3
(4) d = a + c
(5) f = b + c
(6) g = d + f
```

Slicing has many application areas in software engineering, including debugging, verification, measurement, maintenance... etc. Either performed manually or automatically, the benefits of slicing in areas mentioned above are obvious. In case of performing a task manually, it is clear that by removing irrelevant information the user can focus on the relevant aspects of the problem and therefore work more quickly and accurately. Moreover, in case of performing the task automatically, the required time for completing the computing can be greatly improved if we manage to remove the irrelevant data from the computation.

1.2 Types of Slicing

Slicing techniques can be classified in many forms depending on the chosen criteria. Moreover, we can distinguish slicing the source code, as illustrated in the previous section, from the slicing of execution traces as our tool Julienne does. One of the most common classifications also distinguish different forms of slicing such as the static, dynamic or conditional slicing.

Static slicing is the type of slicing in which we do not consider any particular execution of the program, that is, we slice a program with independence of the input data. The opposite of static slicing is the dynamic slicing in which we consider the possible values of the input data. Finally, conditional slicing fills the gap between static and dynamic slicing and "preserves the

semantics of the slicing criterion only for those inputs that satisfy a boolean condition" [17]. The following is an example showing the differences of static and dynamic slicing. Given the following source code and the **input value of 10 for the variable** `a`:

```
(1) load(a)
(2) if (a < 0)
(3)     a = 0
(4) b = a + 1
```

The result of applying static slicing with the slicing criterion specified as `b` yields the sliced program:

```
(1) load(a)
(2) if (a < 0)
(3)     a = 0
(4) b = a + 1
```

Whereas the result of applying dynamic slicing with the same slicing criterion is as follows:

```
(1) load(a)
(2) if (a < 0)
(3)     a = 0
(4) b = a + 1
```

The difference, as previously mentioned, is that the dynamic slicing detects that lines 2 and 3 of the source code do not affect the value of the provided slicing criterion **for the given input value of** `a`, and therefore can be safely removed.

1.2.1 Backward Trace Slicing

Backward trace slicing is the type of slicing that consists of performing the slicing of an execution trace that is processed from back to front using a slicing criterion that is fixed on the final state of the trace.

This project focuses on the Julienne, which applies the backward trace slicing technique for conditional rewrite theories [1] [4] by slicing a given execution trace with respect to a slicing criterion that is specified in any selected state of the trace, not only the final one. The possibility of fixing the slicing criterion in any state of the trace, except for the first one, is one of the many new features developed in this project. This technique traces back

relevant data by means of an incremental technique of matching refinement that is much more lighter than previous backward tracing approaches [1].

A very detailed approach to this technique, including definitions and examples can be found in [4].

1.2.2 Incremental Slicing

Incremental slicing refers to the possibility to ask for multiple consecutive applications of a particular slicing technique in order to achieve a more accurate slicing. In any application of the selected slicing technique, the slicing criterion can either be redefined completely or simply refined to achieve better results.

Our new version of JULIENNE supports incremental slicing by allowing the user to slice a provided execution trace any number of times by specifying a slicing criterion in any state, except for the first state, up to the state in which the previous slicing was performed. An example of an incremental slicingn, performed using JULIENNE, can be found in Section 4.2.

CHAPTER 2

JULIENNE Online Trace Slicer

JULIENNE [1] is an online trace slicer tool developed by members of the *Extensions of Logic Programming (ELP)* research group at the *Department of Information System and Computation* inside the *Universitat Politècnica de València*. The first version of this tool allowed to apply the backward-slicing technique [4] in order to slice Maude execution traces. The new and extended version of the tool developed in the current project adds some useful features without neglecting the previous existing ones. The new features can be classified in two major categories, those oriented to ease the task of slicing to the user, that is, improving the usability of the tool, and those adding some advanced functionality. The following sections explain in detail the use, features and implementation of the new tool JULIENNE.

2.1 Providing the Maude Specification

The first step while using JULIENNE is to provide the Maude specification that will be used in the subsequent steps. The user has the choice of either provide a new, custom specification or selecting one of the examples that JULIENNE provides for evaluation purposes, which will be copied automatically in the text input area as it is selected.

2.2 Providing the Execution Trace

After providing the specification, the next step is to provide a valid execution trace. If the user has previously selected one of the given examples as input specification, there will be one execution trace already loaded that, of course, can be replaced if desired. If not, there are two ways to provide it. The first way is to copy a valid (i.e., with no syntax errors and related to the previously provided specification) execution trace. The second way is use the trace generator with which JULIENNE is endowed by specifying two valid states, initial and final, of the desired execution trace. JULIENNE trace

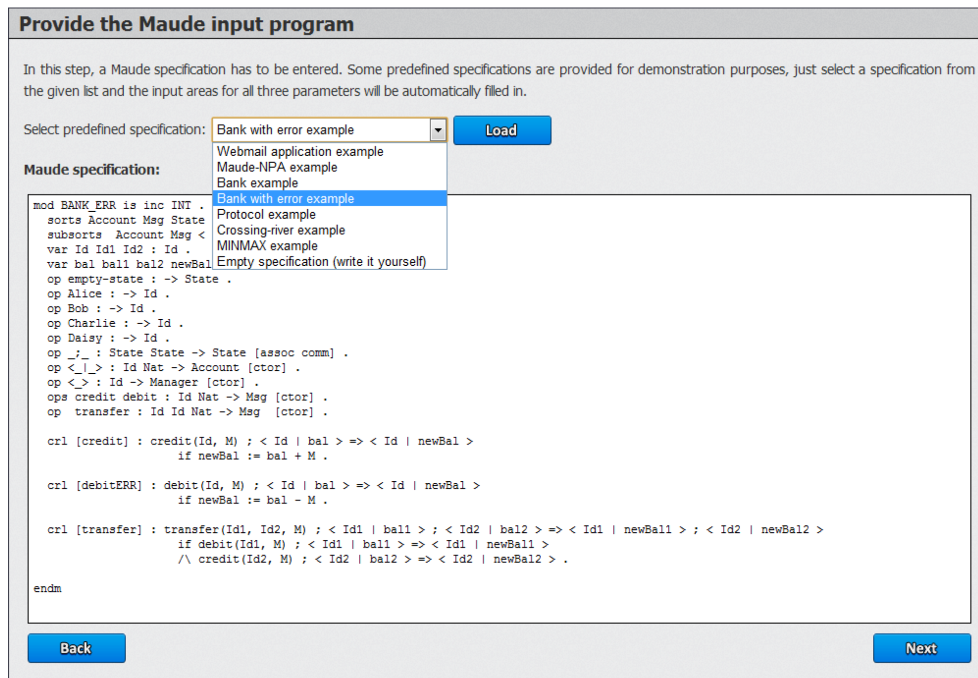

Figure 2.1: JULIENNE showing a list of predefined examples and loading the *Bank with error example* specification.

generator will find, if it is possible, one execution trace that leads from the given initial state to the final one. This last way of obtaining an execution trace by means of the trace generator can be done by giving the states either at the meta-level or source-level representation.

At this point, an error can arise because of a mistake in the provided specification, provided states, execution trace, or because there is no path leading from the given initial state to the final one. In case of error, the corresponding input text area will flash in red and a proper message will alert the user as shown in Figure 2.4.

2.3 Navigating through the Trace

Once the user has provided a valid specification and execution trace, JULIENNE will show in a slider all the states of the trace, from the first to the last one, offering the possibility of viewing the information also in source-level or meta-level representation. This way, the user can visit any state by using the navigation buttons or by specifying a particular state and pressing

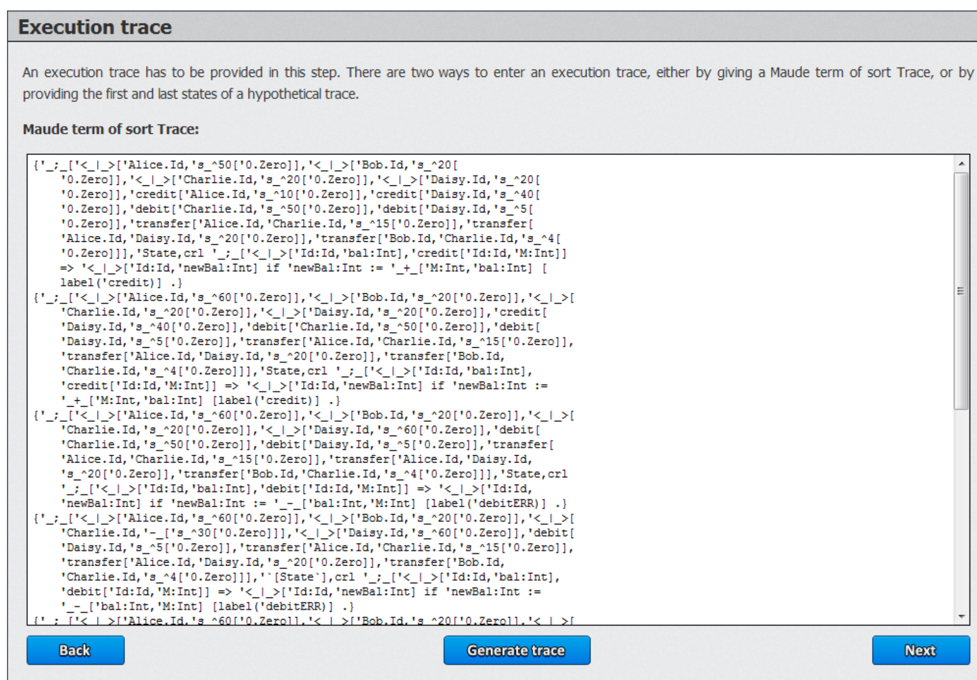

Figure 2.2: Maude execution trace loaded in Julienne.

the *Go* button. Then, the user can select the slicing criterion in any of the states, not only in the last state.

In addition, by pressing the arrow that symbolizes the transition between states, Julienne displays useful information about that particular transition, as shown in Figure 2.5.

2.4 Specifying the Slicing Criterion

The next step in our methodology is to specify the desired slicing criterion. This task can be performed by highlighting with the mouse any part of the considered relevant text of any state with the exception, of course, of the first one, as it does not have any previous state and the slicing is performed backwards. The user can select as many criteria as she wants in any of the states but only those selected in the earliest state will be considered as the slicing criterion. For example, if Julienne shows an execution trace of 22 states like the one in Figure 2.6 and the user selects some text in states 5, 10 and 20, the slicing criterion will be retrieved from the 5th state.

Also, while selecting the slicing criterion, the user can left-click any se-

Figure 2.3: Generation of a Maude execution trace by providing both the initial and final state.

lected text to remove the highlighting or just select the *Clear slicing criterion* button, which will remove all the slicing criteria previously specified.

2.5 Performing the Slice

After specifying the slicing criterion, the backward-slicing technique is applied and the delivered sliced trace will be displayed together with the original trace up to the state where the slicing process started to be applied. That means that, if the execution trace has 22 states and the user selects a slicing criterion in the 20th state, the new trace will only have the first 20 states. However, if the user feels that the criterion was useless or wants to specify a different one, just by selecting the *Restore original trace* button, the original trace will be fully restored as expected.

At this point, the user has the possibility of either restore the full original trace and perform another different slice or slice even more the current sliced trace in the same way the original trace was sliced to remove further unnecessary information. This is what we refer with incremental slicing and

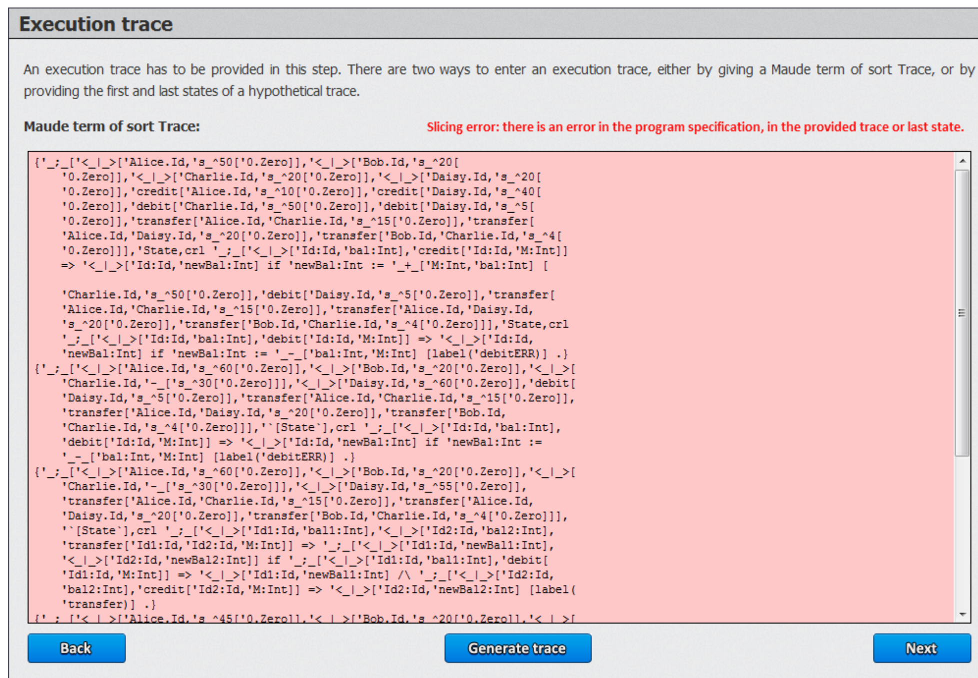

Figure 2.4: JULIENNE flashing in red and alerting the user about a possible mistake in the provided information.

will be shown in more detail in Section 4.2.

Additionally, another useful feature that is only available when a sliced trace is displayed is the possibility of partially hide irrelevant information by means of the *Hide irrelevant data* checkbox.

2.6 Trace Slice Results

Each time a slice is performed, by selecting the *Show trace slice* button, the user can access to a table that contains all the information of the slicing process, that is, the number of steps, the rule applied at each step, the original trace and the sliced trace. This information can be displayed in either source-level or meta-level representation and, in both representations, the irrelevant information can be partially hidden by means of the *Hide irrelevant data* checkbox, a feature that is also available while navigating the sliced trace.

Rule:

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

Substitution:

Id1 / Bob
Id2 / Charlie
M / 4
bal1 / 20
bal2 / - 15
newBal1 / 16
newBal2 / - 11

Position:

Lambda . 2

Figure 2.5: Information displayed (in source-level representation) after pressing state transition arrow.

Slicing criterion

Set the data that you want to trace back along the execution trace. The slicing criterion is typically given as a set of interesting positions of the last term of the trace. In order to facilitate the selection of the relevant information to be traced back, Julienne provides a simple mouse-based user interface so that you could select the data that you want to trace back by highlighting directly the relevant information. In addition, it is possible to navigate over the states and redefine the slicing criterion in any state of the trace in order to take advantage of the even more simplification for debugging purposes.

Show advanced view

These are the states of your Trace: States 20-21 of 22

< Alice | 25 > ; < Daisy | 75 > ; < Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4) → < Alice | 25 > ; < Daisy | 75 > ; < Bob | 16 > ; < Charlie | - 11 >

#state Go

Back | Restore original trace | Clear slicing criterion | Show trace slice | Slice

Figure 2.6: Slicing criteria fixed in the states 20 and 21 of the execution trace.

Slicing criterion

Set the data that you want to trace back along the execution trace. The slicing criterion is typically given as a set of interesting positions of the last term of the trace. In order to facilitate the selection of the relevant information to be traced back, Julienne provides a simple mouse-based user interface so that you could select the data that you want to trace back by highlighting directly the relevant information. In addition, it is possible to navigate over the states and redefine the slicing criterion in any state of the trace in order to take advantage of the even more simplification for debugging purposes.

Show advanced view   Hide irrelevant data

These are the states of your Trace: States 19-20 of 20

```
< Alice | 25 > ; < Bob | 20 > ; < Charlie | - 15 > ; < Daisy | 75 > ; transfer(Bob,Charlie,4)
→ < Alice | 25 > ; < Daisy | 75 > ; < Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4)
```

These are the states of your Sliced Trace:

```
* ; * ; < * | 75 > ; < * | 20 > ; transfer(*,*,*)
***deleted***
```

#state Go

Back | Restore original trace | Clear slicing criterion | Show trace slice | Slice

Figure 2.7: Result obtained after applying the backward-slicing technique with the slicing criterion described in Figure 2.6.

Chapter 3

Implementation

3.1 Technologies

In this section we briefly comment the different technologies that we have used to develop Julienne.

3.1.1 Maude

Maude is a high-performance reflective language and system created by José Meseguer at the *Stanford Research Institute International* (SRI International) and licensed under the terms of the GNU General Public License. The official version is currently being developed by an international team [8] of researchers, although any user can access and modify its source code. Usually, the modifications are given in the form of separate modules which can be included by means of a key word. Some of the principal characteristics of Maude described in [16] are the following:

- Based on rewriting logic.
- Wide-spectrum.
- Multiparadigm.
- Reflective.
- Internal Strategies.

In our tool, Maude plays a key role since it is the language we also used to implement the backward-slicing technique described in [4], as well as generate the state maps and the encoded messages that contain the results to be delivered from the Maude system to Julienne.

The first version of our tool had all the developed Maude code contained in an ad hoc module named *ConditionalSlicing*. This second version of the tool slightly modifies this module and has all the new developed code in a new

different module named *Julienne*, which also includes the *ConditionalSlicing* module, as it makes use of it.

3.1.2 HyperText Mark-up Language

The *HyperText Mark-up Language* (HTML) is a mark-up language created in 1990 by Tim Berners-Lee at the *Conseil Européen pour la Recherche Nucléaire* (CERN), shortly after establishing [14] the bases for the *HyperText Transfer Protocol* (HTTP) and therefore the *World Wide Web*.

In October of 1994, Berners-Lee founded the *World Wide Web Consortium* (W3C), in order to coordinate the efforts to continue developing, maintaining and extending this language. Currently, the W3C foundation is responsible for developing and publishing the standards of this language. In January of 2008 the last version of the standard, HTML5, was published as a working draft and it is expected to be fully developed by 2014, although nowadays almost all browsers support it entirely.

Being an online tool, JULIENNE is developed using HTML in collaboration with other languages. In particular, HTML was used to create the bases of the interface, whose appearance and behaviour where modified later by means of CSS and JavaScript, respectively. We focused on extending a previous version of the tool as well as adapting it to the last version of the HTML standard, HTML5. The primary reasons to such decision were to offer the same experience to the user regardless of the browser that is being used and to extend the life of the code as much as possible.

3.1.3 Cascading Style Sheets

Cascading Style Sheets (CSS) is a style sheet language created [15] in 1994 by Håkon Wium Lie, also at CERN. The primary purpose of this language is to cover a gap intentionally left by HTML with respect to styling documents in order to achieve different visual representations of a document without modifying it and only associating a style sheet.

Being closely related to HTML, CSS standards are developed and published by the same organization as HTML, the W3C. At the present time, the last fully developed version available is CSS3, although since September 29, 2009, a new draft for CSS4 was started. However, unlike the HTML5 standard, which is fully supported by all major browsers despite not being completely developed, CSS4 is not yet supported by any of them, so we chose to use the CSS3 standard in our tool.

In JULIENNE, CSS3 was used to configure the appearance of all elements of the GUI, that is, specifying their size, font, color, position, margins, ...

etc. This was achieved following two different strategies. The first strategy is useful to specify the style information of highly customized elements and consists in specifying all the desired rules directly for the element. The second strategy is useful when different elements share the same values of attributes; for example background color or image, line widths, ... etc. In order to specify the value of those attributes, instead of repeating each rule for each element, CSS3 allows all the desired rules to be grouped within a class and then associate the class to any number of elements. An HTML element can have many associated classes and thus benefit from all of them. In case of conflict, that is, an element has associated two classes which specify different values for the same attribute, the browser will determine the most specific rule and apply it. Furthermore, this second strategy is ideal for changing the visual appearance of elements dynamically, that is, we can associate and detach previously specified classes to an element by means of JavaScript in response to any user interaction.

3.1.4 JavaScript

JavaScript is a scripting language formalized in the ECMAScript language standardized by *Ecma International - European Association for Standardizing Information and Communication Systems*, founded in 1961, in the ECMA-262 specification and ISO/IEC 16262. Currently, the most recent revision of the ECMA-262 standard specification is v. 5.1, published in June 2011.

Despite of its name, JavaScript is not an evolution from the Java language, although it is influenced by it in terms of naming conventions. In fact, JavaScript is influenced by languages C, Java, Perl and Python and even though it has its utility outside the web environment, it is there where shows its full potential.

In Julienne, JavaScript is used client-side to implement all the behaviour of the tool except for the calculations related to the application of the backward-slicing technique and generations of state maps. This behaviour ranges from the very simple task of highlighting one button when the mouse is over it to the most complicated task of decryption of a state map and the subsequent calculus of the position of the slicing criterion. The following list contains some significant features of the tool that were implemented by using JavaScript:

- The slider displaying the states of an execution trace.
- All the animations between steps in the tool.

- Display of errors.
- Recovery of the specified slicing criterion.
- Calculus of the positions of the terms included in the slicing criterion.
- Hiding or showing of irrelevant data.
- Change of the displayed information format depending on whether normal or advanced mode is enabled.
- Display of the relevant information of the transitions between states.

3.2 Implementation Details

In this section we give a detailed review about the main insight of the implementation. We summarize the most relevant characteristics and argue the implementation decisions taken.

3.2.1 Graphical User Interface

Let us start by describing the front-end of the tool.

3.2.1.1 General Decisions

When this project was started, the first aspect that was addressed was improving the visual appearance of JULIENNE. This allowed us to have a first contact with some of its code and somehow become familiar with it. In this aspect, we decided to change the previous white background with one in a darkest gray tone in order avoid eye strain in case of prolonged exposure. Also, some minor details like aligns, position of elements and text font and size were altered just for designing purposes. All these changes were made by modifying the HTML and CSS code, which was also reorganized and updated to the latest available standard by avoiding the use of deprecated elements. This decision was more important than it firstly appears, as each browser can behave differently to the same non-standardized or outdated code and we wanted to offer the same experience to all users regardless of the browsers they were using to access JULIENNE.

After those changes were made and new buttons and checkboxes that matched the new design were created, we focused in other major interface changes explained in the following sections.

3.2.1.2 Slider

One of the new features in this version was the ability to navigate through the states of an execution trace. This was achieved by adapting and heavily modifying a very simple JavaScript slider available at http://innominepixel.wordpress.com. Requirements like displaying always two slides containing a pair of states of the execution trace and adding some additional navigation controls were tricky to implement and substantially changed the original code. In addition, as some jQuery animation effects were added just to transmit to the user the feeling of sliding from one state to another, some restrictions were needed in order to disable the navigation controls while an animation was being performed.

Additionally, one of the features previously mentioned in this document, that is, the ability to access the information of each transition, was achieved by adapting the jQuery Popup Window plug-in available at http://www.mywebdeveloperblog.com. Figure 2.5 shows the result of this adaptation.

The entire code of the slider is available in the *slicer.js* file of the tool.

3.2.1.3 Hiding Irrelevant Data

One of the most useful features that JULIENNE offers is the ability of hiding irrelevant data, either in source-level or meta-level representation, both during the navigation of a sliced trace and when accessing the table results. This feature was accomplished through a series of substitutions which follow a similar strategy to the one used when highlighting the slicing criterion. The main idea is to wrap the irrelevant text inside an open and close *span* tags with an associated CSS class that, instead of specifying a background color, specifies a different font color, lighter than the usual.

The corresponding substitutions were made by means of the JavaScript *replace* function, using both literal and regular expressions. The use of regular expressions was mandatory in order to match the structure of the meta-level representation of a state and hide nested irrelevant terms while preserving other terms in higher positions. However, because of the limitations of JavaScript with respect to the regular expression language, some terms had to be protected before starting the recursive replacement task. This is done by first adding specific marks and then deleting those marks after the work was done.

<table>
<tr><th>Step</th><th>RuleName</th><th>Execution trace</th><th>Sliced trace</th></tr>
<tr><td>1</td><td>'Start</td><td>< Alice | 50 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Alice,10) ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>< * | 20 > ; < * | * > ; < * | 20 > ; < * | * > ; credit(*,40) ; credit(*,*) ; debit(*,5) ; debit(*,*) ; transfer(*,*,20) ; transfer(*,*,*) ; transfer(*,*,*)</td></tr>
<tr><td>2</td><td>unflattening</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 50 > ; credit(Alice,10)</td><td>***deleted***</td></tr>
<tr><td>3</td><td>credit</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 60 ></td><td>< * | 20 > ; < * | * > ; < * | * > ; < * | 20 > ; credit(*,40) ; debit(*,5) ; debit(*,*) ; transfer(*,*,20) ; transfer(*,*,*) ; transfer(*,*,*)</td></tr>
<tr><td>4</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>5</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 20 > ; credit(Daisy,40)</td><td>***deleted***</td></tr>
<tr><td>6</td><td>credit</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 60 ></td><td>< * | 20 > ; < * | * > ; < * | 60 > ; < * | * > ; debit(*,5) ; debit(*,*) ; transfer(*,*,20) ; transfer(*,*,*) ; transfer(*,*,*)</td></tr>
<tr><td>7</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 60 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>8</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | 20 > ; debit(Charlie,50)</td><td>***deleted***</td></tr>
<tr><td>9</td><td>debitERR</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | - 30 ></td><td>< * | 20 > ; < * | * > ; < * | * > ; < * | 60 > ; debit(*,5) ; transfer(*,*,20) ; transfer(*,*,*) ; transfer(*,*,*)</td></tr>
<tr><td>10</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>11</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 60 > ; debit(Daisy,5)</td><td>***deleted***</td></tr>
<tr><td>12</td><td>debitERR</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 55 ></td><td>< * | 55 > ; < * | 20 > ; < * | * > ; < * | * > ; transfer(*,*,20) ; transfer(*,*,*) ; transfer(*,*,*)</td></tr>
<tr><td>13</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; < Daisy | 55 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>14</td><td>unflattening</td><td>< Bob | 20 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 60 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15)</td><td>***deleted***</td></tr>
<tr><td>15</td><td>transfer</td><td>< Bob | 20 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 45 > ; < Charlie | - 15 ></td><td>* ; < * | * > ; < * | 55 > ; < * | 20 > ; transfer(*,*,20) ; transfer(*,*,*)</td></tr>
<tr><td>16</td><td>flattening</td><td>< Alice | 45 > ; < Bob | 20 > ; < Charlie | - 15 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>17</td><td>unflattening</td><td>< Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4) ; < Alice | 45 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20)</td><td>***deleted***</td></tr>
<tr><td>18</td><td>transfer</td><td>< Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4) ; < Alice | 25 > ; < Daisy | 75 ></td><td>* ; * ; < * | 75 > ; < * | 20 > ; transfer(*,*,*)</td></tr>
<tr><td>19</td><td>flattening</td><td>< Alice | 25 > ; < Bob | 20 > ; < Charlie | - 15 > ; < Daisy | 75 > ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>20</td><td>unflattening</td><td>< Alice | 25 > ; < Daisy | 75 > ; < Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td colspan="2">Total size:</td><td>3201</td><td>770</td></tr>
<tr><td colspan="4">Reduction: 75%</td></tr>
</table>

Back | Show advanced view | Hide irrelevant data | Compressed view

Figure 3.1: Original versus sliced trace.

3.2.2 Selecting the Slicing Criterion

Older versions of JULIENNE dealt with the problem of selecting the slicing criterion in different ways.

The most simple approach was to let the user specify the positions of

the terms that will be considered as the slicing criterion and then pass them to the backward-slicing operator in Maude. This approach was immediately neglected because of the difficulty that implies for the user to manually count each term in very complicated Maude specifications.

The better approach so far was the use of a pattern-matching language that, executed in Maude, could get the position of the term used as slicing criterion and then perform it. This approach, however, had two major problems. The first problem was the need for the user to provide a valid pattern. This implied not only to know the syntax of the pattern-matching language, but to have some ability to build a pattern that would match the desired term later considered as the slicing criterion. The second problem was that, in some cases, a unique valid pattern could match inevitably more than one term, the desired one and others.

The main idea in all the considered approaches was to ease the task of providing a slicing criterion to the user as much as possible. Therefore, we focused in this aspect and the user is only required to select the data of interest by clicking the mouse, either in the meta-level representation or in the source format of the state.

Because of the way it was implemented (described in the next subsections), this idea had some positive side-effects: Firstly, the simplicity for the user was a major point. Secondly, the user did not need to provide a valid syntax pattern nor selection. Just by selecting one or more characters, all terms with at least one selected character would be part of the slicing criterion. This means that, in this aspect, all the problems resulting from the complexity and lack of knowledge of the specification were solved. Thirdly, the user could easily make multiple-terms and complex criteria. And last but not least, graphically viewing the slicing criterion as highlighted text of the corresponding state has proved to be very helpful to the user.

3.2.2.1 Subterm position

Any execution state of a given Maude specification can be represented as an ordered collection of terms. The structure of this collection can be *easily* identified while reading a meta-level representation of the state, although determining the exact position of each term manually can be very complex depending on the considered Maude specification.

For instance, given the next state in meta-level representation:

```
'_;_['debit['Alice.Id,'s_^30['0.Zero]],'_;_['<_|_>['Alice
.Id,'s_^100['0.Zero]],'credit['Alice.Id,'s_^50['0.Zero]]]]
```

the following Figures (3.2 and 3.3) show the correspondence between the state and its associate position tree in both source-level and meta-level representation.

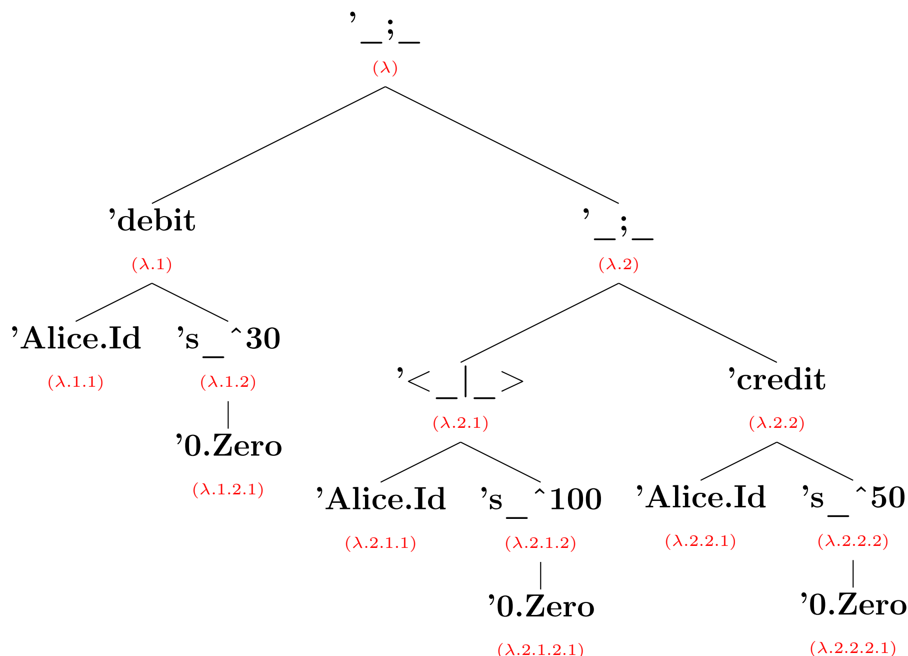

Figure 3.2: Position tree of a state expressed in meta-level representation.

Just by observing the state and its corresponding position tree it is obvious that, starting from the root of the tree, which is addressed by the *Lambda*, the meta-level representation of a tree can be built by traversing the tree using a depth-first left-to-right strategy.

3.2.2.2 Common technical details

The selection of the slicing criterion was implemented by means of an adaptation of a free licensed jQuery plug-in named *jQuery Text Highlighter*. In essence, this plug-in allows us to wrap the selected text with the HTML tag *span*, which **does absolutely nothing**, but whose style can be edited by means of CSS like any other HTML element. For example we can provide a background color, and thus create the effect that the contained text is highlighted. Also, the *span* tags were very distinctive, mostly because they had a custom CSS class specified inside, making it easy to find out later which parts of the text that represented the state were selected (those surrounded by open and close *span* tags), and then calculate the affected terms and their positions.

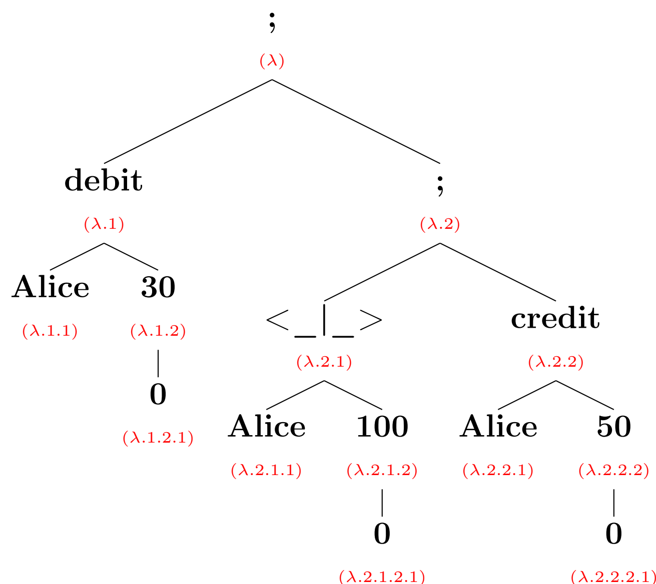

Figure 3.3: Position tree of a state expressed in source-level representation.

For this strategy to work, because of the limitations of the HTML *text area* component, one previous change had to be made. It is impossible to edit the *text area* component so that the text inside can have multiple distinct styles and also insert *span* tags, hence, in order to achieve this effect, the states were shown inside *div* elements, conveniently formatted to achieve the appearance of an original *text area* component.

3.2.2.3 Meta-level representation

As the meta-level representation of any state has a strong, easy to identify, pattern, it is possible to identify the position of, not only a term, but a single character of a term without having any semantic knowledge of it. This is achieved by parsing the state and modifying a variable keeping the current position, starting with *Lambda*, in three different ways, as we find the non-escaped characters "**[**", "**]**" or "**,**".

Finding a "**[**" character means that the current position has to be deepened by one. The opposite effect, that is, to raise one level the current position, has to be made when a non-escaped "**]**" character is found. The last character, "**,**" just increases by one the value of it, as if we travelled to the immediate right branch of the tree of positions representing the state explained in Section 3.2.2.1.

With this strategy, just by parsing the entire state character by character and updating the current position at each step if needed, we can easily find

each relevant position, which are those found while reading between an open *span* tag and its corresponding close tag. The result will be an array of relevant positions to be provided directly to the backward-slicing operator in Maude.

3.2.2.4 Source-level representation

While the meta-level representation had a very distinctive pattern that is common to all possible states, regardless of their specifications, a source-level representation did not have any. This was the main problem to solve, not only in this section, but in the entire new version of the tool.

One of the beauties of Maude is that it provides the means for the user to create complex models whose states can be represented in many, many forms, often limited by his imagination. This makes literally impossible to find a pattern common to all these possible states. Nevertheless, with our approach, this can be easily solved.

At this point, we need to explain that a source-level representation of a state in JULIENNE is the result of a custom Maude operation, *dt*, named for *downTerm*, whose main goal is to lower a given state in meta-level representation from the meta-level to the source-level. This operator *dt* recursively builds the state in source-level representation by translating each term step by step according to the Maude specification provided, and placing them in the correct order.

Once we know how a source-level representation of a state is built, we can easily think that, besides translating each term into its source-level representation, we can translate it into any kind of information that will help us to find the position of any character of the state, that is, we can write the information needed to create a map of the state. Also, with regard to the efficiency of this process, both the calculation of the position of each term and the building of the map can be performed in parallel with the generation of the source-level representation state while using *dt*, as all three functions run throughout the text in the same recursively manner.

With this new information, the formerly very complicated task of finding the positions of the selected terms is greatly eased. First of all, we do not need the information of the state anymore, nor we need to have any semantic knowledge of it. We only need the indices of the selected text inside the string that represents it. The easiest way of obtaining this is to create a mask of the state by replacing any found character, spaces included, with, for example, "**n**" if it is not wrapped inside a *span* tag and "**S**" if it is. Then, we only have to read the map sequentially.

As explained in Section 3.2.2.5, each element of the map contains two

data. The first one describes how many characters the second data will affect. This second data just describes the position of those characters. Once we read an element of the map, we can easily get a substring of the mask starting at the current value of the pointer used to read, and with the length given by the value of the first data of the read element. Then, if this substring contains the character "**S**", the position of the term is included in the result array of positions. At the end, just by building the map, the algorithm that is able to get the positions of the terms is even simpler than the used for meta-level representation.

3.2.2.5 State Map Structure

As mentioned in Section 3.2.2.4, in order to retrieve the slicing criterion specified by the user in the source representation of a state, it is useful to construct a map of it. The fact that JULIENNE allows the slicing criterion to be fixed in any state of the execution trace except for the first one implies that all those states need to be mapped.

Being JULIENNE an online web application, the server hosting our tool needs to deliver this information, along with the two representations of each state, to the user, so the size of all this new generated information needs to be lightened as much as possible, as it is affected by the considered Maude specification and the number of states of the execution trace, which can be huge. The positive side is that this operation only needs to be performed once per execution trace provided, regardless of the number of slices later required by the user.

The minimum information needed to retrieve a position of a term inside a state is both its position according to the string of characters that represents that state and its position according to the tree of positions described in Section 3.2.2.1. For example, given the bank specification described in Section 4.2 and this state in meta-level representation:

```
'_;_['debit['Alice.Id,'s_^30['0.Zero]],'_;_['<_|_>['Alice
.Id,'s_^100['0.Zero]],'credit['Alice.Id,'s_^50['0.Zero]]]]
```

The result of calling the *dt* operator with them as arguments will be:

```
debit(Alice,30) ; < Alice | 100 > ; credit(Alice,50)
```

and the generated map will be:

```
c6p.1c5p.1.1c1p.1c2p.1.2X.1.2.1c1p.1c3pc2p.2.1c5p.2.1.1
c3p.2.1c3p.2.1.2X.2.1.2.1c2p.2.1c3p.2c7p.2.2c5p.2.2.1
c1p.2.2c2p.2.2.2X.2.2.2.1c1p.2.2
```

Let us explain these results. First, the map is a sequence of concatenated elements each consisting of two data components. The first component begins with the `c` character and describes how many characters will affect the second data. This second data begins with the `p` character and describes the position of those characters according with the tree of positions explained in Section 3.2.2.1.

Moreover, it is possible that two terms in meta-level representation will merge into one single joint representation. This happens with the natural numbers, as their meta-level representation comprises two related terms (e.g. `'s_^X['0.Zero]`, being `X` a given number). In order to cover those cases, the second data of an element can describe more than one position and thus need to be delimited, in our case with the `X` character.

Finally, since the root of the tree of positions is always *Lambda*, it is implicit and does not need to be reflected in the map, so an empty second component can be given, meaning that the position is λ itself.

Map element	**Length**	**Characters**	**Positions**
`c6p.1`	6	debit(	$\lambda.1$
`c5p.1.1`	5	Alice	$\lambda.1.1$
`c1p.1`	1	,	$\lambda.1$
`c2p.1.2X.1.2.1`	2	30	$\lambda.1.2$, $\lambda.1.2.1$
`c1p.1`	1	)	$\lambda.1$
`c3p`	3	_;_	λ
`c2p.2.1`	2	<_	$\lambda.2.1$
`c5p.2.1.1`	5	Alice	$\lambda.2.1.1$
`c3p.2.1`	3	_|_	$\lambda.2.1$
`c3p.2.1.2X.2.1.2.1`	3	100	$\lambda.2.1.2$, $\lambda.2.1.2.1$
`c2p.2.1`	2	_>	$\lambda.2.1$
`c3p.2`	3	_;_	$\lambda.2$
`c7p.2.2`	7	credit(	$\lambda.2.2$
`c5p.2.2.1`	5	Alice	$\lambda.2.2.1$
`c1p.2.2`	1	,	$\lambda.2.2$
`c2p.2.2.2X.2.2.2.1`	2	50	$\lambda.2.2.2$, $\lambda.2.2.2.1$
`c1p.2.2`	1	)	$\lambda.2.2$

Table 3.1: Breakdown of a map by elements and their correspondence with the associated state.

With all this in mind, going back to our example, the first element will be `c6p.1`. This means that the first six characters of the source-level repre-

sentation of the state will correspond to the $\lambda.1$ position, that is, `"debit("`. The next element of the map is `c5p.1.1`, which means that the next five characters, `"Alice"`, will be part of the term with position $\lambda.1.1$. And so the map is read sequentially together with the source-level representation of the state.

CHAPTER 4

Using JULIENNE

4.1 Crossing River Example

The simplest example provided for evaluation purposes in the JULIENNE distribution package is the well-known problem of the crossing river. One of the many variants of the statement of this problem is as follows:

> *There is a man, a wolf, a goat and a cabbage in one side of a river that need to cross to the other side by means of a boat. The boat can only accommodate one traveller besides the man, but leaving the wolf and the goat together without his supervision will cause the wolf to eat the goat and so leaving the goat with the cabbage.*

First of all, we have to load the corresponding specification by selecting the crossing-river example as shown in Figure 4.1. The next step is to provide a valid execution trace. In this example, instead of using the default execution trace provided by JULIENNE, we generate a different trace by providing both initial and final states in source-level representation. Our initial state will correspond to the state with all four passengers standing at the left side of the river and our final state will be that with them safely carried to the opposite side. By pressing the *Generate trace* button, JULIENNE will generate one of the possible solutions to this problem and will display it as the provided execution trace.

Now, we press the *Next* button to start slicing. As Figure 4.3 shows, the previously generated execution trace has 56 states. We now chose as the slicing criterion the term of the last state describing the side of the river that the wolf is at, that is, `right`.

After pressing the *Slice* button, the sliced trace states will be displayed along with the original trace states, as Figure 4.4 shows. We can observe that many sliced states are marked with the `***deleted***` text. This means that the state is substantially the same as the preceding one and can be safely omitted.

Finally, by pressing the *Show trace slice* button, we can consult all the information in a condensed table, along with some additional data about the reduction achieved, which in this example is 77% of the original trace (Figure 4.5).

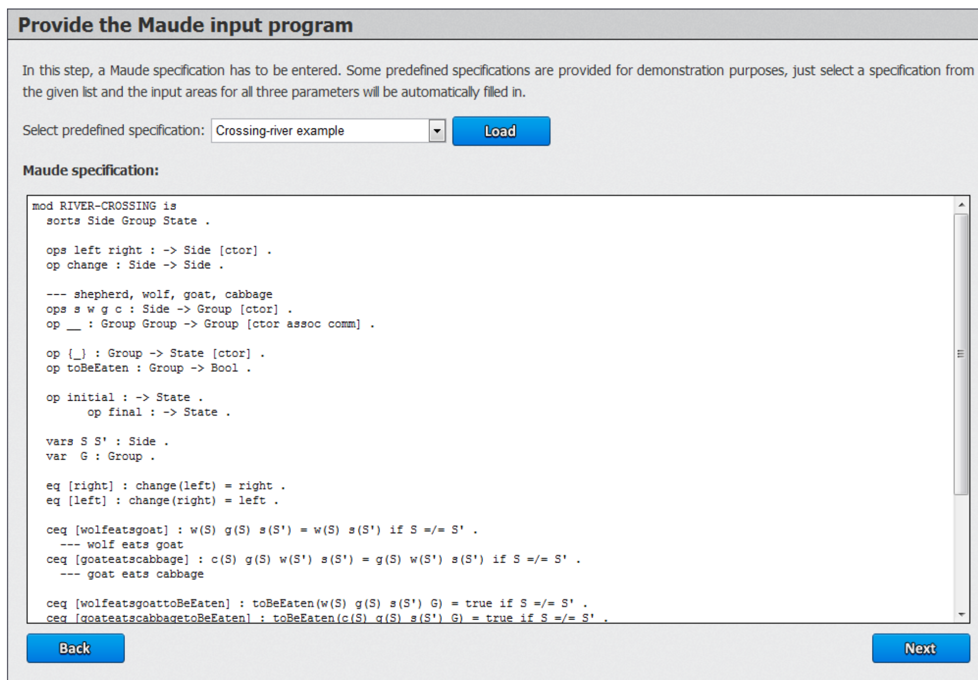

Figure 4.1: JULIENNE showing a specification of the crossing river example.

Figure 4.2: Generation of a Maude execution trace by providing the initial and final state in source-level representation.

Figure 4.3: A Maude execution trace of 56 states with a selected slicing criterion in the last state.

Figure 4.4: Original and sliced trace states after applying the backward-slicing technique with the slicing criterion described in Figure 4.3.

Trace slice

Step	RuleName	Execution trace	Sliced trace
1	'Start	{ c(left) g(left) s(left) w(left) }	{ c(left) g(left) s(left) w(left) }
2	unflattening	{ c(left) w(left) g(left) s(left) }	***deleted***
3	goat	{ s(change(left)) c(left) w(left) g(change(left)) }	{ c(left) g(change(left)) s(change(left)) w(left) }
4	flattening	{ c(left) g(change(left)) s(change(left)) w(left) }	***deleted***
5	unflattening	{ c(left) s(change(left)) w(left) g(change(left)) }	***deleted***
6	right	{ c(left) s(change(left)) w(left) g(right) }	{ c(left) g(right) s(change(left)) w(left) }
7	flattening	{ c(left) g(right) s(change(left)) w(left) }	***deleted***
8	unflattening	{ c(left) g(right) w(left) s(change(left)) }	***deleted***
9	right	{ c(left) g(right) w(left) s(right) }	{ c(left) g(right) s(right) w(left) }
10	flattening	{ c(left) g(right) s(right) w(left) }	***deleted***
11	unflattening	{ c(left) g(right) w(left) s(right) }	***deleted***
12	shepherd-alone	{ c(left) g(right) w(left) s(change(right)) }	{ c(left) g(right) s(change(right)) w(left) }
13	flattening	{ c(left) g(right) s(change(right)) w(left) }	***deleted***
14	unflattening	{ c(left) g(right) w(left) s(change(right)) }	***deleted***
15	left	{ c(left) g(right) w(left) s(left) }	{ c(left) g(right) s(left) w(left) }
16	flattening	{ c(left) g(right) s(left) w(left) }	***deleted***
17	unflattening	{ g(right) w(left) c(left) s(left) }	***deleted***
18	cabbage	{ s(change(left)) g(right) w(left) c(change(left)) }	{ * g(right) s(change(left)) w(left) }
19	flattening	{ c(change(left)) g(right) s(change(left)) w(left) }	***deleted***
20	unflattening	{ g(right) s(change(left)) w(left) c(change(left)) }	***deleted***
21	right	{ g(right) s(change(left)) w(left) c(right) }	***deleted***
22	flattening	{ c(right) g(right) s(change(left)) w(left) }	***deleted***
23	unflattening	{ c(right) g(right) w(left) s(change(left)) }	***deleted***
24	right	{ c(right) g(right) w(left) s(right) }	{ * g(right) s(right) w(left) }
25	flattening	{ c(right) g(right) s(right) w(left) }	***deleted***
26	unflattening	{ c(right) w(left) g(right) s(right) }	***deleted***
27	goat	{ s(change(right)) c(right) w(left) g(change(right)) }	{ * g(change(right)) s(change(right)) w(left) }
28	flattening	{ c(right) g(change(right)) s(change(right)) w(left) }	***deleted***
29	unflattening	{ c(right) s(change(right)) w(left) g(change(right)) }	***deleted***
30	left	{ c(right) s(change(right)) w(left) g(left) }	{ * g(*) s(change(right)) w(left) }
31	flattening	{ c(right) g(left) s(change(right)) w(left) }	***deleted***
32	unflattening	{ c(right) g(left) w(left) s(change(right)) }	***deleted***
33	left	{ c(right) g(left) w(left) s(left) }	{ * g(*) s(left) w(left) }
34	flattening	{ c(right) g(left) s(left) w(left) }	***deleted***
35	unflattening	{ c(right) g(left) s(left) w(left) }	***deleted***
36	wolf	{ s(change(left)) c(right) g(left) w(change(left)) }	{ * * g(*) s(change(left)) }
37	flattening	{ c(right) g(left) s(change(left)) w(change(left)) }	***deleted***
38	unflattening	{ c(right) g(left) w(change(left)) s(change(left)) }	***deleted***
39	right	{ c(right) g(left) w(change(left)) s(right) }	{ * * g(*) s(right) }
40	flattening	{ c(right) g(left) s(right) w(change(left)) }	***deleted***
41	right	{ c(right) g(left) s(right) w(right) }	***deleted***
42	unflattening	{ c(right) g(left) w(right) s(right) }	***deleted***
43	shepherd-alone	{ c(right) g(left) w(right) s(change(right)) }	{ * * g(*) s(change(right)) }
44	flattening	{ c(right) g(left) s(change(right)) w(right) }	***deleted***
45	unflattening	{ c(right) g(left) w(right) s(change(right)) }	***deleted***
46	left	{ c(right) g(left) w(right) s(left) }	{ * * g(*) s(*) }
47	flattening	{ c(right) g(left) s(left) w(right) }	***deleted***
48	unflattening	{ c(right) w(right) g(left) s(left) }	{ * g(*) s(*) }
49	goat	{ s(change(left)) c(right) w(right) g(change(left)) }	{ * * * w(right) }
50	flattening	{ c(right) g(change(left)) s(change(left)) w(right) }	***deleted***
51	unflattening	{ c(right) s(change(left)) w(right) g(change(left)) }	***deleted***
52	right	{ c(right) s(change(left)) w(right) g(right) }	***deleted***
53	flattening	{ c(right) g(right) s(change(left)) w(right) }	***deleted***
54	unflattening	{ c(right) g(right) w(right) s(change(left)) }	***deleted***
55	right	{ c(right) g(right) w(right) s(right) }	***deleted***
56	flattening	{ c(right) g(right) s(right) w(right) }	***deleted***
Total size:		2633	603
Reduction: 77%			

Back   Show advanced view ☐   ☑ Hide irrelevant data   Compressed view

Figure 4.5: Sliced trace showing a reduction of 77% with respect to the original execution trace.

4.2 Bank Example

In this second example, also provided within the JULIENNE distribution package, we work with a very basic Maude specification of a bank system. There are three basic rules, `credit`, `debitERR` and `transfer`. Obviously by its name, the rule labelled as `debitERR` is introduced to illustrate some problems.

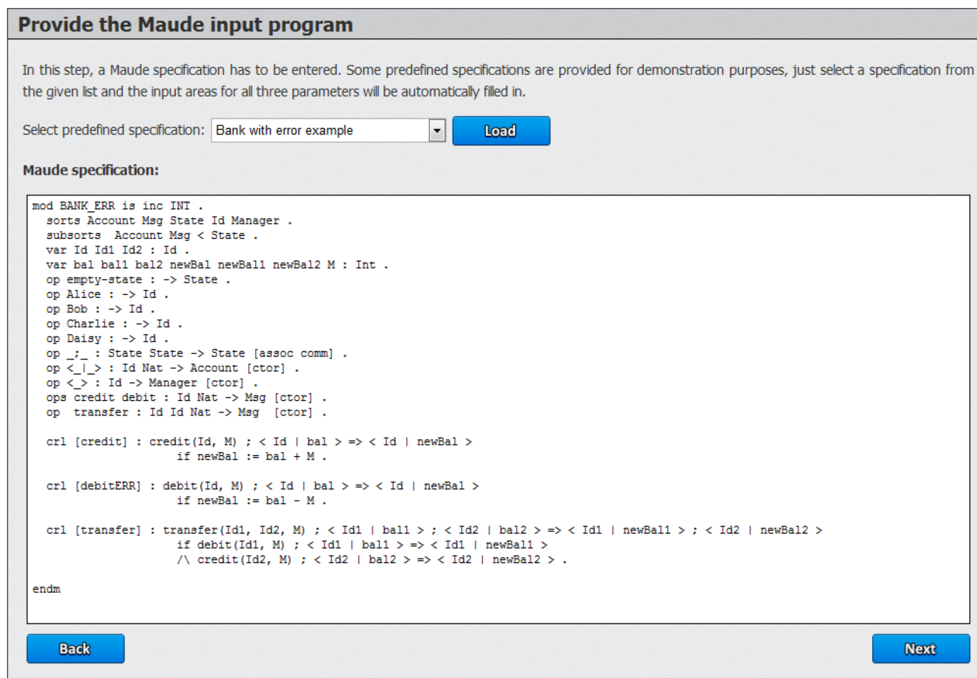

Figure 4.6: JULIENNE showing a Maude bank specification with an error in one of its rules.

The first rule, `credit`, adds some amount of money to a bank account. The second rule, `debitERR` subtracts it, as if we charge the account with any amount of money, and the `transfer` rule just *moves* some money from an account to another one. As we will discover later, the problem is that the `debitERR` does not check if there is enough money to be withdrawn from the given account.

Once we load the specification, we press the *Next* button and get a default execution trace also provided by JULIENNE. As this trace is affected by the error we want to discover, we just press *Next* again to move forward to the trace navigation step.

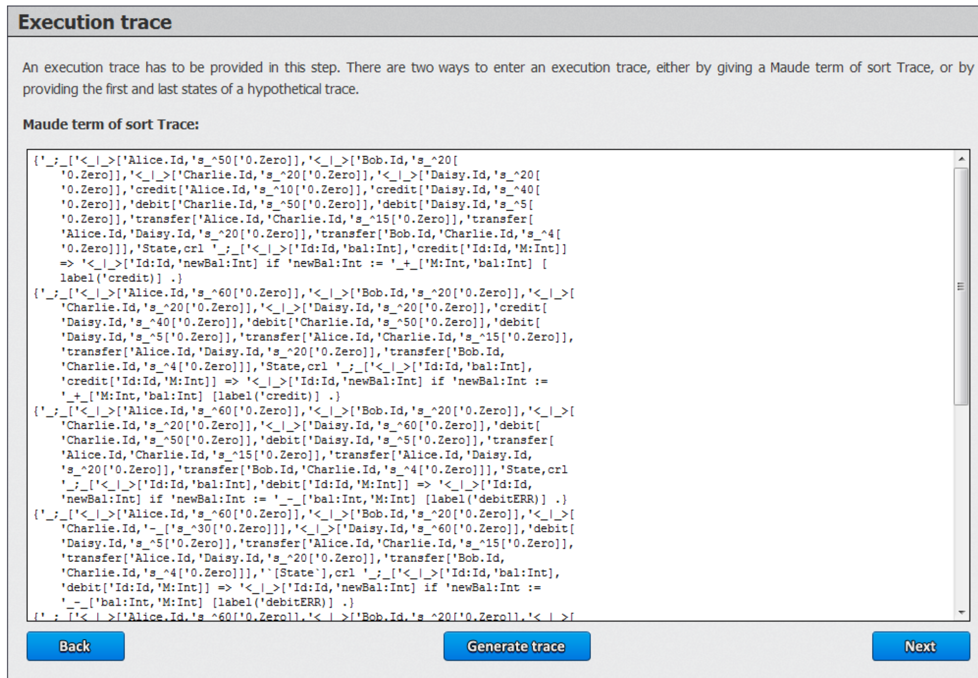

Figure 4.7: Execution trace provided as an example for the faulty Bank specification.

After we reach this step, we can clearly observe that the given execution trace has 22 states and the last one contains a term that describes a bank account, whose holder is Charlie, with negative balance. In order to discover the error that causes such a negative balance, we specify as the slicing criterion the minus symbol and then slice the trace by pressing the *Slice* button.

Following the first application of the backward-slicing technique the sliced trace will appear and we will be able to consult all the information in the available table by pressing the *Show trace slice* button. It shows that a reduction of 75% has been achieved with respect to the original trace.

Also, by hiding irrelevant information with the *Hide irrelevant data* option, we can clearly see that the first time we have a negative balance in an account is in state number 9, so the rule applied in the transition from the 8th to 9th states will almost certainly hold the error. Not surprisingly, by inspecting the information available about the transition, we find out that the rule applied is `debitERR`.

Now that we know which transition is holding the error, we can go to the 9th state and perform another slice in order to further reduce the sliced

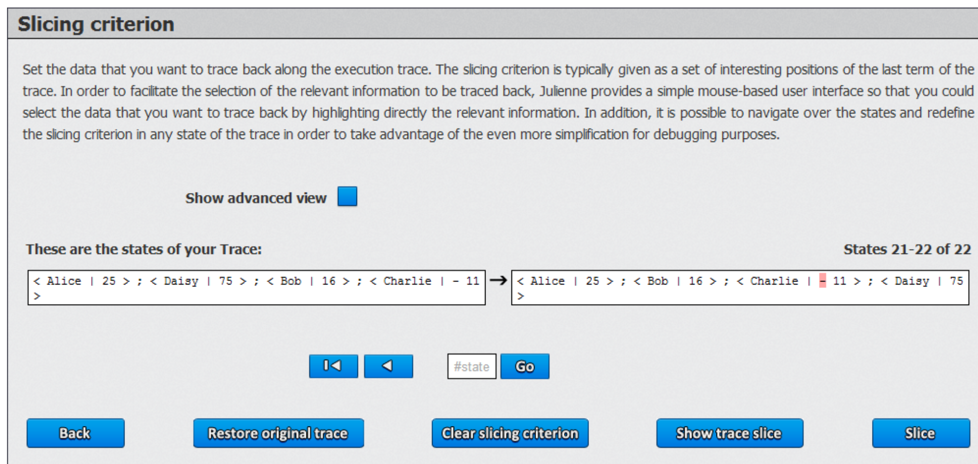

Figure 4.8: A 22 state execution trace of the faulty Bank specification with the minus symbol selected as the slicing criterion.

trace. We can now chose `"- 30"` as the slicing criterion and, after applying the backward-slicing technique again, we obtain the sliced trace described in Figures 4.12 and 4.13.

If we compare both table results of Figures 4.10 and 4.13, we can notice that, with the first application of the technique, the relevant information in the first state of the sliced trace were the account initial balance, the debit operation and two transfer operations. Those transfer operations do not appear as relevant in the resulting slice of the second application of the technique because, in fact, they are not relevant, as they affect the final balance of the account because their rules are applied after the *debitERR* is.

To conclude, we can state that this example demonstrates that by slicing a second time at an early state we can achieve more specific sliced traces and thus smaller.

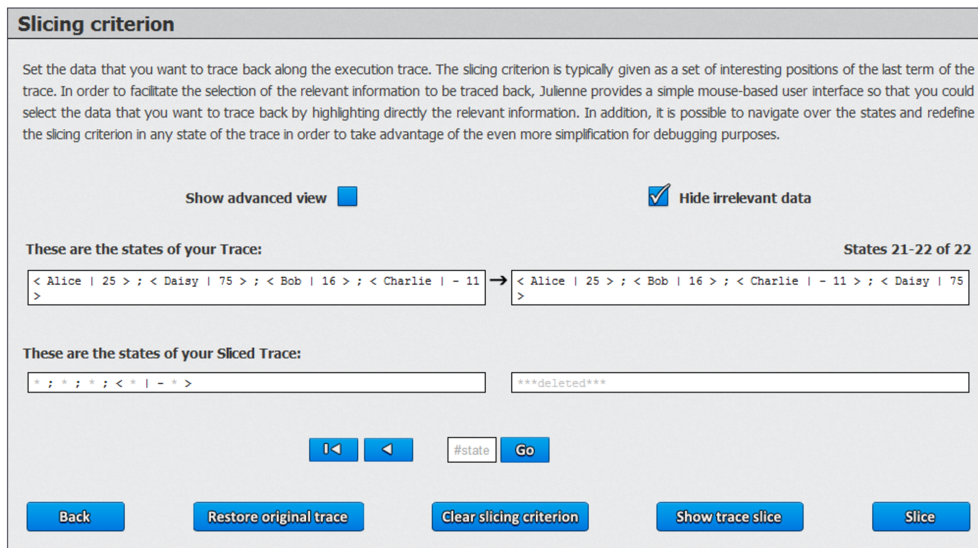

Figure 4.9: Snapshot of the tool after applying the backward-slicing technique in Figure 4.8.

Trace slice

<table>
<tr><th>Step</th><th>RuleName</th><th>Execution trace</th><th>Sliced trace</th></tr>
<tr><td>1</td><td>'Start</td><td>< Alice | 50 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Alice,10) ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>< * | * > ; < * | 20 > ; < * | * > ; < * | * > ; credit(*,*) ; credit(*,*) ; debit(*,*) ; debit(*,50) ; transfer(*,*,4) ; transfer(*,*,*) ; transfer(*,*,15)</td></tr>
<tr><td>2</td><td>unflattening</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 50 > ; credit(Alice,10)</td><td>***deleted***</td></tr>
<tr><td>3</td><td>credit</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 60 ></td><td>< * | * > ; < * | * > ; < * | 20 > ; < * | * > ; credit(*,*) ; debit(*,*) ; debit(*,50) ; transfer(*,*,4) ; transfer(*,*,*) ; transfer(*,*,15)</td></tr>
<tr><td>4</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>5</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 20 > ; credit(Daisy,40)</td><td>***deleted***</td></tr>
<tr><td>6</td><td>credit</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 60 ></td><td>< * | * > ; < * | * > ; < * | * > ; < * | 20 > ; debit(*,*) ; debit(*,50) ; transfer(*,*,4) ; transfer(*,*,*) ; transfer(*,*,15)</td></tr>
<tr><td>7</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 60 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>8</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | 20 > ; debit(Charlie,50)</td><td>***deleted***</td></tr>
<tr><td>9</td><td>debitERR</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | - 30 ></td><td>< * | * > ; < * | * > ; < * | - 30 > ; < * | * > ; debit(*,*) ; transfer(*,*,4) ; transfer(*,*,*) ; transfer(*,*,15)</td></tr>
<tr><td>10</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>11</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 60 > ; debit(Daisy,5)</td><td>***deleted***</td></tr>
<tr><td>12</td><td>debitERR</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 55 ></td><td>< * | * > ; < * | * > ; < * | * > ; < * | - 30 > ; transfer(*,*,4) ; transfer(*,*,*) ; transfer(*,*,15)</td></tr>
<tr><td>13</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | - 30 > ; < Daisy | 55 > ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>14</td><td>unflattening</td><td>< Bob | 20 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 60 > ; < Charlie | - 30 > ; transfer(Alice,Charlie,15)</td><td>***deleted***</td></tr>
<tr><td>15</td><td>transfer</td><td>< Bob | 20 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 45 > ; < Charlie | - 15 ></td><td>< * | * > ; < * | - 15 > ; < * | * > ; < * | * > ; transfer(*,*,4) ; transfer(*,*,*)</td></tr>
<tr><td>16</td><td>flattening</td><td>< Alice | 45 > ; < Bob | 20 > ; < Charlie | - 15 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>17</td><td>unflattening</td><td>< Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4) ; < Alice | 45 > ; < Daisy | 55 > ; transfer(Alice,Daisy,20)</td><td>***deleted***</td></tr>
<tr><td>18</td><td>transfer</td><td>< Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4) ; < Alice | 25 > ; < Daisy | 75 ></td><td>* ; * ; < * | * > ; < * | - 15 > ; transfer(*,*,4)</td></tr>
<tr><td>19</td><td>flattening</td><td>< Alice | 25 > ; < Bob | 20 > ; < Charlie | - 15 > ; < Daisy | 75 > ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>20</td><td>unflattening</td><td>< Alice | 25 > ; < Daisy | 75 > ; < Bob | 20 > ; < Charlie | - 15 > ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>21</td><td>transfer</td><td>< Alice | 25 > ; < Daisy | 75 > ; < Bob | 16 > ; < Charlie | - 11 ></td><td>* ; * ; * ; < * | - * ></td></tr>
<tr><td>22</td><td>flattening</td><td>< Alice | 25 > ; < Bob | 16 > ; < Charlie | - 11 > ; < Daisy | 75 ></td><td>***deleted***</td></tr>
<tr><td colspan="2">Total size:</td><td>3337</td><td>807</td></tr>
<tr><td colspan="4">Reduction: 75%</td></tr>
</table>

Back | Show advanced view | Hide irrelevant data | Compressed view

Figure 4.10: Original versus sliced trace for the Bank example.

Figure 4.11: States 8 and 9 of the execution trace displayed in Figure 4.9 with a slicing criterion specified in state 9.

Figure 4.12: Snapshot of JULIENNE after applying the backward-slicing technique in Figure 4.11.

Trace slice

<table>
<tr><th>Step</th><th>RuleName</th><th>Execution trace</th><th>Sliced trace</th></tr>
<tr><td>1</td><td>'Start</td><td>< Alice | 50 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Alice,10) ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>* ; * ; * ; * ; * ; < * | * > ; < * | 20 > ; < * | * > ; credit(*,*) ; credit(*,*) ; debit(*,50)</td></tr>
<tr><td>2</td><td>unflattening</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 50 > ; credit(Alice,10)</td><td>***deleted***</td></tr>
<tr><td>3</td><td>credit</td><td>< Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Alice | 60 ></td><td>* ; * ; * ; * ; * ; * ; < * | * > ; < * | 20 > ; credit(*,*) ; debit(*,50)</td></tr>
<tr><td>4</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 20 > ; credit(Daisy,40) ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>5</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 20 > ; credit(Daisy,40)</td><td>***deleted***</td></tr>
<tr><td>6</td><td>credit</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Daisy | 60 ></td><td>* ; * ; * ; * ; * ; * ; * ; < * | 20 > ; debit(*,50)</td></tr>
<tr><td>7</td><td>flattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Charlie | 20 > ; < Daisy | 60 > ; debit(Charlie,50) ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4)</td><td>***deleted***</td></tr>
<tr><td>8</td><td>unflattening</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | 20 > ; debit(Charlie,50)</td><td>* ; < * | 20 > ; debit(*,50)</td></tr>
<tr><td>9</td><td>debitERR</td><td>< Alice | 60 > ; < Bob | 20 > ; < Daisy | 60 > ; debit(Daisy,5) ; transfer(Alice,Charlie,15) ; transfer(Alice,Daisy,20) ; transfer(Bob,Charlie,4) ; < Charlie | - 30 ></td><td>* ; < * | - 30 ></td></tr>
<tr><td colspan="2">Total size:</td><td>1772</td><td>267</td></tr>
<tr><td colspan="4">Reduction: 84%</td></tr>
</table>

Back | Show advanced view | Hide irrelevant data | Compressed view

Figure 4.13: Table of results of Figure 4.12.

Conclusions and Future Work

In this project, we have developed a tool for slicing execution traces delivered by Maude. In order to achieve better reduction rates in the sliced execution traces, apart from the chosen slicing technique that we apply, we focus on aspects that are crucial for the slicing effectiveness: to provide a suitable slicing criterion and the state of the trace in which the slicing technique will be applied.

To improve the accuracy of the slicing criterion, which is typically provided by the user, we endow JULIENNE with a graphical facility that simplifies the process of identifying and fixing it. Moreover, by allowing the slicing technique to be applied in any state of the provided execution trace, JULIENNE offers to the user the possibility to apply the slicing technique in an incremental way and even change the slicing criterion for each application. The new version of JULIENNE offers solutions to both problems and provides the user with a very friendly working environment to get the best possible results.

As possible future work, we plan to integrate JULIENNE with an incremental debugger for Maude which uses backward trace slicing to improve the debugging experience.

Bibliography

[1] Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Julienne: a Trace Slicer for Conditional Rewrite Theories. In: Proc. FM 2012. Springer LNCS 7436: 28–32 (2012)

[2] Alpuente, M., Ballis, D., Espert, J., Romero, D.: Model-checking Web Applications with Web-TLR. In: Proc. ATVA 2010. Springer LNCS 6252: 341–346 (2010)

[3] Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward Trace Slicing for Rewriting Logic Theories. In: Proc. CADE 2011. Springer LNCS 6803: 34–48 (2011)

[4] Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Backward Trace Slicing for Conditional Rewrite Theories. In: Proc. LPAR-18. Springer LNCS 7180: 62–76 (2012)

[5] Alpuente, M., Ballis, D., Romero, D.: Specification and Verification of Web Applications in Rewriting Logic. In: Proc. FM 2009. Springer LNCS 5850: 790–805 (2009)

[6] Baggi, M., Ballis, D., Falaschi, M.: Quantitative Pathway Logic for Computational Biology. In: Proc. CMSB '09. Springer LNCS 5688: 68–82 (2009)

[7] Chen, F., Rosu, G.: Parametric Trace Slicing and Monitoring. In: TACAS. Springer LNCS 5505: 246–261 (2009)

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.6):

[9] Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.C.: The Maude Formal Tool Environment. In: Proc. CALCO. Springer LNCS 4624: 173–178 (2007)

[10] The JULIENNE Web site (2012), available at: `http://users.dsic.upv.es/grupos/elp/soft.html`

[11] The JULIENNE V. 2 Web site (2012), available at: http://safe-tools.dsic.upv.es/julienneV2

[12] TeReSe (ed.): Term Rewriting Systems. Cambridge University Press, Cambridge, UK (2003)

[13] Martí-Oliet, N., Palomino, M., Verdejo, A.: Rewriting logic bibliography by topic: 1990-2011. Journal of Logic and Algebraic Programming. To appear (2012)

[14] Berners-Lee, T. Information Management: A Proposal.

[15] Lie, H. W. Cascading HTML Style Sheets: A Proposal.

[16] Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude. Electronic Notes in Theoretical Computer Science, Vol. 4: 65–89 (1996)

[17] Gallagher, K., Binkley, D.: Program Slicing. Advances in Computers, Vol. 43: 1–50 (1996)

[18] Silva, J.: A Vocabulary of Program Slicing-Based Techniques. In: ACM Comput. Surv. 44, 3. Article 12 (June 2012).