Slicing-Based Trace Analysis of Rewriting Logic Specifications with *i*JULIENNE*

María Alpuente[1], Demis Ballis[2], Francisco Frechina[1], and Julia Sapiña[1]

[1] DSIC-ELP, Universitat Politècnica de València,
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
{alpuente,ffrechina,jsapina}@dsic.upv.es
[2] DIMI, Università degli Studi di Udine,
Via delle Scienze 206, 33100 Udine, Italy
demis.ballis@uniud.it

Abstract. We present *i*JULIENNE, a trace analyzer for conditional rewriting logic theories that can be used to compute abstract views of Maude executions that help users understand and debug programs. Given a Maude execution trace and a slicing criterion which consists of a set of target symbols occurring in a selected state of the trace, *i*JULIENNE is able to track back reverse dependences and causality along the trace in order to incrementally generate highly reduced program and trace slices that reconstruct all and only those pieces of information that are needed to deliver the symbols of interest. *i*JULIENNE is also endowed with a trace querying mechanism that increases flexibility and reduction power and allows program runs to be examined at the appropriate level of abstraction.

1 Introduction

Execution traces are an important source of information for program understanding and debugging. Standard tracers usually present execution histories that mainly consist of low-level execution steps so that the relationship between the executed program and the execution history is not easy to derive because some key dependences that are naturally expressed at the programming language level can be either scattered or omitted in the trace. This is particularly true for those systems that are specified in Rewriting Logic (RWL) —a logic of change that can deal naturally with highly nondeterministic concurrent computations.

Rewriting logic is efficiently implemented in the high-performance language Maude. Execution traces generated by Maude are complex objects to deal with. The traces typically include thousands of rewrite steps that are obtained by applying the equations and rules of the considered specification (including all the internal rewrite steps for evaluating the conditions of such equations/rules).

* This work has been partially supported by the EU (FEDER) and the Spanish MEC project ref. TIN2010-21062-C02-02, and Generalitat Valenciana ref. PROMETEO2011/052, and was carried out during the tenure of D. Ballis' ERCIM "Alain Bensoussan" Postdoctoral Fellowship. The research leading to these results has received funding from the EU 7^{th} Framework Programme (FP7/2007-2013) under agreement n. 246016. F. Frechina is supported by FPU-ME grant AP2010-5681.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 121–124, 2013.

In addition, Maude traces are incomplete because algebraic axiom applications, which implicitly occur in an equational simplification process that is hidden within Maude's *matching modulo* algorithm, are not recorded at all in the trace. This provides a very low-level blueprint of program execution whose manual inspection is frequently unfeasible or, in the best case, is an extremely labor-intensive and time-consuming task.

This paper describes *i*Julienne, a slicing-based trace analysis tool that assists the user in the comprehension and debugging of RWL theories that are encoded in Maude. *i*Julienne is built on top of a trace slicer that implements the backward conditional trace slicing algorithm described in [2,3,4]. Roughly speaking, the trace slicing mechanism included in *i*Julienne rolls back the program execution (making *all* the rewrite and equational simplification steps explicit) while tracking back only and all data in the trace that are needed to accomplish the selected *slicing criterion* —that is, the data that contribute to producing the set of *target symbols* that occur in the observed state of the trace. The core trace slicer included within *i*Julienne is a totally redesigned implementation of our slicing technique in [2,3] that supersedes and greatly improves the preliminary system presented in [4]. In particular, the new trace analyzer *i*Julienne is equipped with an *incremental* backward trace slicing algorithm that supports stepwise refinements of the trace slice and achieves huge reductions in the size of the trace. Starting from a Maude execution trace $\mathcal{T}$, a slicing criterion $\mathcal{S}$ can be attached to any given state of the trace and the computed trace slice $\mathcal{T}^\star$ can be repeatedly refined by applying backward trace slicing w.r.t. increasingly restrictive versions of $\mathcal{S}$. Furthermore, the system supports a cogent form of *dynamic program slicing* [7] as follows. Given a Maude program $\mathcal{M}$ and a trace slice $\mathcal{T}^\star$ for $\mathcal{M}$, *i*Julienne is able to infer the minimal fragment of $\mathcal{M}$ (i.e., the *program slice*) that is needed to reproduce $\mathcal{T}^\star$. Finally, *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 each state transition (and its corresponding sliced counterpart) at different granularity levels.

2 *i*Julienne at Work

The *i*Julienne system is written in Maude and consists of about 250 Maude function definitions. It can be invoked as a Maude command or used online through a Java Web service. The tool is publicly available at [6] together with several case studies which consider large execution traces, such as the counter-examples delivered by the Maude LTLR model-checker [1]. A thorough experimental evaluation of our slicing methodology can be found in [5].

To illustrate how *i*Julienne works in practice, we show a typical trace slicing session on a Maude implementation of *Blocks World* —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

```
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
```

Fig. 1. `BLOCKS-WORLD` faulty Maude specification

goal of the robot arm is to produce one or more vertical stacks of blocks. In our specification, which is shown in the Maude module `BLOCKS-WORLD` of Figure 1, we define a Blocks World system with three different kinds of blocks that are defined by means of the operators `a`, `b`, and `c` of sort `Block`. Different blocks have different sizes that are described by using the unary operator `size`. We also consider some operators that formalize block and robot arm properties whose intuitive meanings are given in the accompanying program comments.

The states of the system are modeled by means of associative and commutative lists of properties of the form $\texttt{prop}_1\&\texttt{prop}_2\&\ldots\&\texttt{prop}_n$, which describe any possible configuration of the blocks as well as the status of the robot arm. The system behavior is formalized by four, simple rewrite rules that control the robot arm. Specifically, the `pickup` rule describes how the robot arm grabs a block from the table, while `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 $\texttt{B}_1$ from being piled on a block $\texttt{B}_2$ if the size of $\texttt{B}_1$ is greater than the size of $\texttt{B}_2$.

Barely perceptible, the Maude specification of Figure 1 fails to provide a correct Blocks World implementation. By using the `BLOCKS-WORLD` module, it is indeed possible to derive system states that represent erroneous configurations. For instance, the initial state

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

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

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

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 $\mathcal{T} = \mathtt{s_i} \rightarrow^* \mathtt{s_f}$, and we initially slice $\mathcal{T}$ w.r.t. the slicing criterion that observes the two anomalous occurrences of the `empty` property and the stack `on(a,b)` 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. Alternatively, we can also query the trace using an appropriate pattern, which extracts the considered target data by means of pattern-matching, to State $\mathtt{s_f}$. *i*Julienne yields a trace slice which only records 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. This allows us to deduce that the malfunction is located in one or more rules and equations that are included in the computed program slice.

The generated trace slice is then browsed backwards using the *i*Julienne's navigation facility in search of a possible explanation for the wrong behavior. During this phase, we found an inconsistent state that models a robot arm that is holding block `a` and is empty at the same time. Therefore, we further refine the trace slice by incrementally applying backward trace slicing to the detected, inconsistent state w.r.t. the slicing criterion `hold(a)`. This way, we achieve a trace reduction of ∼90% in which we can easily observe that `hold(a)` only depends on the `clear(a)` and `table(a)` properties. Furthermore, the computed program slice includes the single `pickup` rule. Thus, we can conclude that: (i) the malfunction is certainly located in the `pickup` rule (since the computed program slice only contains that rule); (ii) the `pickup` rule does not depend on the status of the robot arm (this is witnessed by the fact that `hold(a)` only relies on the `clear(a)` and `table(a)` properties); (iii) by (i) and (ii), we can deduce that the `pickup` rule is incorrect, as it never checks the emptiness of the robot arm before grasping a block.

References

1. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Model-Checking Web Applications with Web-TLR. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 341–346. Springer, Heidelberg (2010)
2. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward Trace Slicing for Rewriting Logic Theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 34–48. Springer, Heidelberg (2011)
3. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Backward Trace Slicing for Conditional Rewrite Theories. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 62–76. Springer, Heidelberg (2012)
4. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Julienne: A Trace Slicer for Conditional Rewrite Theories. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 28–32. Springer, Heidelberg (2012)
5. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Using Conditional Trace Slicing for Improving Maude Programs. Science of Comp. Progr. (to appear, 2013)
6. The *i*julienne website (2013), `http://safe-tools.dsic.upv.es/iJulienne`
7. Korel, B., Laski, J.: Dynamic Program Slicing. Inf. Process. Lett. 29(3), 155–163 (1988)