Equational Abstractions in Rewriting Logic and Maude

Narciso Martí-Oliet[1(✉)], Francisco Durán[2], and Alberto Verdejo[1]

[1] Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
narciso@ucm.es
[2] E.T.S.I. Informática, Universidad de Málaga, Málaga, Spain

Abstract. Maude is a high-level language and high-performance system supporting both equational and rewriting computation for a wide range of applications. Maude also provides a model checker for linear temporal logic. The model-checking procedure can be used to prove properties when the set of states reachable from an initial state in a system is finite; when this is not the case, it may be possible to use an equational abstraction technique for reducing the size of the state space. Abstraction reduces the problem of whether an infinite state system satisfies a temporal logic property to model checking that property on a finite state abstract version of the original infinite system. The most common abstractions are quotients of the original system. We present a simple method for defining quotient abstractions by means of equations identifying states. Our method yields the minimal quotient system together with a set of proof obligations that guarantee its executability, which can be discharged with tools such as those available in the Maude formal environment. The proposed method will be illustrated by means of detailed examples.

Keywords: Maude · Rewriting logic · Model checking · Abstraction · Formal environment

1 Introduction

Given a concurrent system, we want to check whether certain properties hold in it or not. If the number of reachable states is *finite*, one can use model checking; however, if the number of such states is *infinite* (or just too large), model checking does not work. For these systems, we can calculate an *abstract* version of the infinite-state transition system, with a finite set of states, to which model checking can be applied. A simple method for defining an abstraction is by means of a *quotient* that collapses the set of states [5].

In the rewriting logic framework implemented in Maude [1], a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where Σ is a signature

Research supported by MINECO Spanish projects StrongSoft (TIN2012–39391–C04–04) and TIN2011–23795, and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).

C. Braga and N. Martí-Oliet (Eds.): SBMF 2014, LNCS 8941, pp. 17–31, 2015.
DOI: 10.1007/978-3-319-15075-8_2

declaring types and operations, E is a set of equations, and R is a set of rules. The quotient abstraction is specified by a set of equations E, added to $\mathcal{R}$, resulting in a rewrite theory $\mathcal{R} = (\Sigma, E \cup E, R)$. However, such a quotient will only be useful, for model-checking purposes, if $\mathcal{R}$ is executable, as detailed later, and the state predicates are preserved by equations [5]. These *proof obligations* (executability and state predicate preservation) can be discharged using tools in the Maude Formal Environment [3].

This paper has the three following goals:

1. To introduce Maude as a framework for modeling systems and model checking their properties.
2. To present a simple method of defining quotient abstractions by means of equations collapsing the set of states.
3. To show how the Maude Formal Environment tools can help in discharging the associated proof obligations.

All of this is going to be done by means of examples. The theoretical basis for the work summarized here has already been described in previous papers, where the reader can find all the missing details [2,3,5].

The following section introduces two examples; in the first, the set of reachable states is finite and model checking will be applied to get the desired results, while in the second this will not be possible because the set of reachable states is infinite. Section 3 first summarizes the concepts necessary and then introduces the equational abstraction method and the associated proof obligations. In Section 4 we apply in detail the method to the second example and manage to get an abstract version satisfying all the requirements, so that we can model check on it the desired property.

2 Maude by Example

In order to model a system in rewriting logic, that is, to specify such a system in Maude, we distinguish between its static part (state structure) and its dynamics (state transitions). The static part is specified as an equational theory, while the dynamics are specified by means of rules. Computation in a transition system is then precisely captured by the term rewriting relation using those rules, where terms represent states of the given system. Moreover, rules need only specify the part of the system that actually changes, so that the frame problem is avoided.

This distinction is reflected in Maude by the difference between functional and system modules [1]. Functional modules in Maude correspond to equational theories (Σ, E) which are assumed to be Church-Rosser (confluent and sort decreasing) and terminating; their operational semantics is equational simplification, that is, rewriting of terms until a canonical form is obtained. Equations are used to define functions over static data as well as properties of states. Usually the equations E are divided into a set A of structural axioms (such as associativity, commutativity, or identity), also known as equational attributes, for which matching algorithms exist in Maude, and a set E' of equations that are Church-Rosser and terminating modulo A.

System modules in Maude correspond to rewrite theories $(\Sigma, A \cup E', R)$; rewriting with R is performed modulo the equations $A \cup E'$. Furthermore, the rules R must be coherent with the equations E' modulo A [2], allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. By assuming coherence, Maude always reduces to canonical form using E before applying any rule in R.

Next we illustrate the application of these general ideas to two different examples.

2.1 Crossing the River

In our first example, we consider a famous puzzle where a shepherd needs to transport to the other side of a river a wild dog, a lamb, and a cabbage. He has only a boat with room for the shepherd himself and another item. The problem is that in the absence of the shepherd the wild dog would eat the lamb, and the lamb would eat the cabbage.

We represent the shepherd and his belongings as objects[1] with only an attribute indicating its river side location. The group is put together by means of an associative and commutative juxtaposition operation. Constants `left` and `right` represent the two sides of the river. Operation `ch`(ange) is used to modify the corresponding attributes. Finally, the rules represent the ways of crossing the river that are allowed by the small capacity of the boat. For instance, the rule labeled `wdog`, for wild dog, specifies that when the shepherd and the wild dog are on the same side of the river they can cross together.

```
mod RIVER-CROSSING is
  sorts Side Group .
  ops left right : -> Side [ctor] .
  op ch : Side -> Side .
  eq ch(left) = right .
  eq ch(right) = left .
  ops s w l c : Side -> Group [ctor] .
  op __ : Group Group -> Group [ctor assoc comm] .
  var S : Side .
  rl [shepherd] : s(S) => s(ch(S)) .
  rl [wdog] : s(S) w(S) => s(ch(S)) w(ch(S)) .
  rl [lamb] : s(S) l(S) => s(ch(S)) l(ch(S)) .
  rl [cabbage] : s(S) c(S) => s(ch(S)) c(ch(S)) .
endm
```

In Section 2.4 we will see how to solve the puzzle, that is, how to find a way of crossing the river satisfying all the constraints and without having the possibility of losing any item in the process, by means of the Maude model checker.

[1] Although Maude has a specific notation for objects, we do not make use of it in this example.

2.2 An Unordered Communication Channel

For our second example, consider a communication channel in which messages can get out of order. There is a sender and a receiver. The sender is sending a sequence of data items, for example numbers. The receiver is supposed to obtain the data items in the same order they were sent. To achieve this in-order communication in spite of the unordered nature of the channel, the sender sends each data item in a message together with a sequence number. The receiver sends back an acknowledgement indicating that the item has been received.

Sequences are specified as lists, while the contents of the unordered channel are modeled as a multiset of messages of sort `Conf`(iguration) using the appropriate equational attributes. The entire system state is a 5-tuple of sort `State`, built by means of the operator `{_,_|_|_,_}` in the module below, where the components are: a buffer with the items to be sent, a counter for the acknowledged items, the contents of the unordered channel, a buffer with the items received, and a counter for the items received.[2]

```
fmod UNORDERED-CHANNEL-EQ is
  sorts Nats List Msg Conf State .
  op 0 : -> Nats [ctor] .
  op s : Nats -> Nats [ctor] .
  op nil : -> List [ctor] .
  op _;_ : Nats List -> List [ctor] .  *** list cons
  op _@_ : List List -> List .         *** list append
  op [_,_] : Nats Nats -> Msg [ctor] .
  op ack : Nats -> Msg [ctor] .
  subsort Msg < Conf .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
  op {_,_|_|_,_} : List Nats Conf List Nats -> State [ctor] .
  vars N : Nats .        vars L P : List .
  eq nil @ L = L .
  eq (N ; L) @ P = N ; (L @ P) .
endfm
```

Having defined all the necessary infrastructure in the previous funcional module, the following system module adds the rules modeling the transitions sending and receiving messages. For instance, the rule labeled `rec` specifies that a message `[N, J]` in the channel is read by the receiver, which adds the data `N` at the end of its sequence, increments its counter to `s(J)`, and puts the corresponding acknowledgement `ack(J)` in the channel.

```
mod UNORDERED-CHANNEL is
  including UNORDERED-CHANNEL-EQ .
  vars N M J : Nats .     vars L P : List .      var  C : Conf .
```

[2] Maude provides predefined modules for natural numbers, lists, and many other datatypes, but they cannot be used in this specification because they are not compatible with most tools in the Maude Formal Environment.

```
  rl [snd]: { N ; L, M | C | P, J } => { N ; L, M | [N, M] C | P, J } .
  rl [rec]: { L, M | [N, J] C | P, J }
        => { L, M | ack(J) C | P @ (N ; nil), s(J) } .
  rl [rec-ack]: { N ; L, J | ack(J) C | P, M } => { L, s(J) | C | P, M } .
endm
```

At the end of Section 4 we will manage to model check that the intended property is indeed satisfied by going through an appropriate quotient specified by a set of equations.

2.3 The Maude Formal Environment

The Maude Formal Environment [3] provides several tools for proving essential properties of Maude modules:

- Maude Termination Tool (MTT) to prove termination of equations and of rules in modules by connecting to external termination tools (we use the AProVe tool [4] below).
- Church-Rosser Checker (CRC) to check the Church-Rosser property of equational specifications.
- Sufficient Completeness Checker (SCC) to check that defined functions have been fully defined in terms of constructors.
- Coherence Checker (ChC) to check the coherence between rules and equations in system modules.
- Inductive Theorem Prover (ITP) to verify inductive properties of functional modules (we will not make use of this tool in our examples).

To show how these tools are used, we apply them to the system module `UNORDERED-CHANNEL` introduced above. First, we check termination of the equational part.

```
Maude> (select tool MTT .)
The MTT has been set as current tool.

Maude> (select external tool aprove .)
aprove is now the current external tool.

Maude> (ct UNORDERED-CHANNEL .)
Success: The module UNORDERED-CHANNEL is terminating.
```

Second, we check that the equational part is also Church-Rosser, which depends on its termination (if the specification has no unjoinable critical pairs, then it is locally confluent; if it is in addition terminating, then it is confluent [2]). The `submit` command, which submits all pending proof obligations to the corresponding tools, makes the connection between the proofs.

```
Maude> (select tool CRC .)
The CRC has been set as current tool.
```

```
Maude> (ccr UNORDERED-CHANNEL .)
Church-Rosser check for UNORDERED-CHANNEL
All critical pairs have been joined.
The specification is locally-confluent.
The module is sort-decreasing.

Maude> (submit .)
The termination goal for the functional part of UNORDERED-CHANNEL has
    been submitted to MTT.
The functional part of module UNORDERED-CHANNEL has been checked
    terminating.
Success: The module is therefore Church-Rosser.
Success: The module UNORDERED-CHANNEL is Church-Rosser.
```

Third, we check that the equational part is sufficiently complete, which depends on it being also terminating and Church-Rosser.

```
Maude> (select tool SCC .)
The SCC has been set as current tool.

Maude> (scc UNORDERED-CHANNEL .)
Sufficient completeness check for UNORDERED-CHANNEL
Completeness counter-examples: none were found
Freeness counter-examples: none were found
Analysis: it is complete and it is sound
Ground weak termination: not proved
Ground sort-decreasingness: not proved

Maude> (submit .)
The sort-decreasingness goal for UNORDERED-CHANNEL has been submitted
    to CRC.
The termination goal for the functional part of UNORDERED-CHANNEL has
    been submitted to MTT.
Church-Rosser check for UNORDERED-CHANNEL
    The module is sort-decreasing.
Success: The functional module UNORDERED-CHANNEL is sufficiently
    complete and has free constructors.
```

Finally, we check that the rules are coherent with respect to the equations, and this depends on all the previous checks.

```
Maude> (select tool ChC .)
The ChC has been set as current tool.

Maude> (cch UNORDERED-CHANNEL .)
Coherence checking of UNORDERED-CHANNEL
All critical pairs have been rewritten and no rewrite with rules can
happen at non-overlapping positions of equations left-hand sides.
The sufficient-completeness, termination and Church-Rosser properties
must still be checked.
```

```
Maude> (submit .)
The Church-Rosser goal for UNORDERED-CHANNEL has been submitted to CRC.
The Sufficient-Completeness goal for UNORDERED-CHANNEL has been
    submitted to SCC.
The termination goal for the functional part of UNORDERED-CHANNEL has
    been submitted to MTT.
Sufficient completeness check for UNORDERED-CHANNEL   [...]
Church-Rosser check for UNORDERED-CHANNEL  [...]
The functional part of module UNORDERED-CHANNEL has been checked
    terminating.
The module UNORDERED-CHANNEL has been checked Church-Rosser.
Success: The module UNORDERED-CHANNEL is coherent.
```

2.4 Model Checking

Temporal logic allows the specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens), related to the possibly infinite global behavior of a system. Maude includes a model checker to prove properties expressed in linear temporal logic (LTL) [1].

The semantics of temporal logic is defined on Kripke structures, which are triples $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set of states, $\rightarrow_{\mathcal{A}}$ is a total binary relation on A representing the state transitions, and $L : A \longrightarrow \mathcal{P}(AP)$ is a labeling function associating to each state $a \in A$ the set $L(a)$ of those atomic propositions in AP that hold in a.

Given a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, one chooses a type k in `M` as the type of states (this is done in the module below by means of a subsort declaration) and extends the module by declaring some state properties Π (of type `Prop`) and defining their meaning by means of additional equations using the basic "satisfaction operator"

```
op _|=_ : State Prop -> Bool .
```

Section 3 below details how then a Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow^{1}_{\mathcal{R}})^{\bullet}, L_{\Pi})$ is obtained. The relation $\mathcal{K}(\mathcal{R}, k)_{\Pi}, t \models \varphi$, where φ is a linear temporal formula and t is the initial state, can be model checked under a few assumptions about the module `M` and its extension with the properties, including the one stating that the set of states reachable from t is finite.

In the crossing-the-river example, the state type is `Group` and we define the following two basic properties:

- `success` characterizes the (good) state in which the shepherd and his belongings have all crossed the river; if we assume that in the initial state all of them are on the `left` side, in the final state all of them are on the `right` side.
- `disaster` characterizes the (bad) states in which some eating takes place, because the shepherd is on the other side.

```
mod RIVER-CROSSING-PROP is
  protecting RIVER-CROSSING .
  including MODEL-CHECKER .
  subsort Group < State .
  op initial : -> Group .
  eq initial = s(left) w(left) l(left) c(left) .
  ops disaster success : -> Prop [ctor] .
  vars S S' S'' : Side .
  ceq (w(S) l(S) s(S') c(S'') |= disaster) = true if S =/= S' .
  ceq (w(S'') l(S) s(S') c(S) |= disaster) = true if S =/= S' .
  eq (s(right) w(right) l(right) c(right) |= success) = true .
  eq G:Group |= P:Prop = false [owise] .
endm
```

Since the model checker only returns either `true` or paths that are counterexamples of properties, in order to find a solution to the puzzle, that is, to find a safe path in the river crossing example, we need a formula that expresses the negation of the property we want: a counterexample will then witness a safe path for the shepherd. If no safe path exists, then it is true that whenever `success` is reached, a disastrous state has been traversed before. The following LTL formula specifies this implication:

```
  <> success -> ((~ success) U disaster)
```

A counterexample to this temporal logic formula (or any other equivalent formula) is a safe path, completed so as to have a cycle.

```
Maude> red modelCheck(initial, <> success -> ((~ success) U disaster)) .
result ModelCheckResult: counterexample(
    {s(left) w(left) l(left) c(left),'lamb}
    {s(right) w(left) l(right) c(left),'shepherd}
    {s(left) w(left) l(right) c(left),'wdog}
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'cabbage}
    {s(right) w(right) l(left) c(right),'shepherd}
    {s(left) w(right) l(left) c(right),'lamb}
    {s(right) w(right) l(right) c(right),'lamb}
    {s(left) w(right) l(left) c(right),'shepherd}
    {s(right) w(right) l(left) c(right),'wdog}
    {s(left) w(left) l(left) c(right),'lamb}
    {s(right) w(left) l(right) c(right),'cabbage}
    {s(left) w(left) l(right) c(left),'wdog},
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'lamb})
```

The path described by the first eight lines in this answer to our model checking request provides the solution that we wanted for the crossing-the-river puzzle.

3 Equational Abstractions

The unordered channel example cannot be model checked directly because the space of reachable states is infinite, since the first rule may be repeatedly applied, sending multiple copies of each message into the channel. It requires thus the application of the abstraction technique in order to be model checked. We summarize here the basic concepts necessary to understand our equational abstraction method [5].

An AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ between Kripke structures $\mathcal{A}$ and $\mathcal{B}$ over the same set AP of atomic propositions is a total relation $H \subseteq A \times B$ such that, when $a \rightarrow_{\mathcal{A}} a'$ and aHb, then there is $b' \in B$ with $a'Hb'$ and $b \rightarrow_{\mathcal{B}} b'$, and, furthermore, if aHb then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$. The simulation H is strict when the previous inclusion is indeed an equality.

A simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ reflects the satisfaction of a formula φ if $\mathcal{B}, b \models \varphi$ and aHb implies $\mathcal{A}, a \models \varphi$.

Theorem 1. *[5] AP-simulations reflect satisfaction of $LTL^-(AP)$ formulas (where $LTL^-(AP)$ is the negation-free fragment of LTL).*

Strict simulations reflect satisfaction of $LTL(AP)$ formulas.

Often we only have a Kripke structure $\mathcal{M}$ and a surjective function to a set of abstract states $h : M \longrightarrow A$. The minimal system $\mathcal{M}^h_{\min}$ (over A) corresponding to $\mathcal{M}$ and h is defined by $(A, \rightarrow_{\mathcal{M}^h_{\min}}, L_{\mathcal{M}^h_{\min}})$, where:

- $x \rightarrow_{\mathcal{M}^h_{\min}} y \iff \exists a.\exists b.(h(a) = x \wedge h(b) = y \wedge a \rightarrow_{\mathcal{M}} b)$
- $L_{\mathcal{M}^h_{\min}}(a) = \bigcap_{x \in h^{-1}(a)} L_{\mathcal{M}}(x)$.

Theorem 2. *[5] $h : \mathcal{M} \longrightarrow \mathcal{M}^h_{\min}$ is indeed a simulation.*

Minimal systems can also be seen as quotients. For a Kripke structure $\mathcal{A}$ and $\sim$ an equivalence relation on A, define $\mathcal{A}/\sim = (A/\sim, \rightarrow_{\mathcal{A}/\sim}, L_{\mathcal{A}/\sim})$, where:

- $[a_1] \rightarrow_{\mathcal{A}/\sim} [a_2] \iff \exists a'_1 \in [a_1].\ \exists a'_2 \in [a_2].\ a'_1 \rightarrow_{\mathcal{A}} a'_2$
- $L_{\mathcal{A}/\sim}([a]) = \bigcap_{x \in [a]} L_{\mathcal{A}}(x)$.

Theorem 3. *[5] Given $\mathcal{M}$ and h surjective, the Kripke structures $\mathcal{M}^h_{\min}$ and $\mathcal{M}/\sim_h$ are isomorphic, where $x \sim_h y$ iff $h(x) = h(y)$.*

The adjective minimal is appropriate since $\mathcal{M}^h_{\min}$ is the most accurate approximation to $\mathcal{M}$ consistent with h, but it is not always possible to have a computable description of $\mathcal{M}^h_{\min}$ because the transition relation:

$$x \rightarrow_{\mathcal{M}^h_{\min}} y \iff \exists a.\exists b.(h(a) = x \wedge h(b) = y \wedge a \rightarrow_{\mathcal{M}} b)$$

is not recursive in general. Here we present methods that, when successful, yield a computable description of $\mathcal{M}^h_{\min}$. As explained before, a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ which determines, for each type k, a transition system $(T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet)$ where

- $T_{\Sigma/E,k}$ is the set of equivalence classes $[t]$ of terms of type k, modulo the equations E;
- $(\rightarrow^1_{\mathcal{R}})^\bullet$ completes the one-step rewrite relation $\rightarrow^1_{\mathcal{R}}$ with an identity pair $([t],[t])$ for each deadlock state $[t]$, to get a total relation.

LTL properties are associated to $\mathcal{R}$ and a type k by specifying the basic state predicates Π in an equational theory $(\Sigma', E \cup D)$ extending (Σ, E) conservatively. State properties are constructed with operators $p : s_1 \dots s_n \rightarrow \mathit{Prop}$ and their semantics is defined by means of equations D using the basic "satisfaction operator" $\_ \models \_ : k\ \mathit{Prop} \rightarrow \mathit{Bool}$. A state property $p(u_1, \dots, u_n)$ holds in a state $[t]$ iff

$$E \cup D \vdash \quad t \models p(u_1, \dots, u_n) = \mathit{true}.$$

The Kripke structure associated to $\mathcal{R}$, k, and Π, with atomic propositions

$$AP_\Pi = \{p(u_1, \dots, u_n) \text{ ground} \mid p \in \Pi\}$$

is then defined as $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi)$ where

$$L_\Pi([t]) = \{p(u_1, \dots, u_n) \mid p(u_1, \dots, u_n) \text{ holds in } [t]\}.$$

Assuming that the equations $E \cup D$ are Church-Rosser and terminating, and that the rewrite theory $\mathcal{R}$ is executable, the resulting Kripke structure is indeed computable.

We can define an abstraction for $\mathcal{K}(\mathcal{R}, k)_\Pi$ by specifying an equational theory extension $(\Sigma, E) \subseteq (\Sigma, E \cup E')$ which gives rise to an equivalence relation $\equiv_{E'}$ on $T_{\Sigma/E}$

$$[t]_E \equiv_{E'} [t']_E \iff E \cup E' \vdash t = t' \iff [t]_{E \cup E'} = [t']_{E \cup E'}$$

and therefore a quotient abstraction $\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'}$. We then need to answer the following question: Is $\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'}$ the Kripke structure associated to another rewrite theory?

We focus on those rewrite theories $\mathcal{R}$ satisfying the following requirements:

- $\mathcal{R}$ is k-deadlock free, that is $(\rightarrow^1_{\mathcal{R}})^\bullet = \rightarrow^1_{\mathcal{R}}$ on $T_{\Sigma/E,k}$,
- $\mathcal{R}$ is k-topmost, so k only appears as the coarity of a certain operator $f : k_1 \dots k_n \longrightarrow k$, and
- no terms of type k appear in the conditions.

A rewrite theory $\mathcal{R}$ can often be transformed into an equivalent one satisfying these requirements [5]. In particular, the unordered channel example satisfies these requirements.

Let us take a closer look at the quotient:

$$\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'} = (T_{\Sigma/E,k} / \equiv_{E'}, (\rightarrow^1_{\mathcal{R}})^\bullet / \equiv_{E'}, L_{\Pi/\equiv_{E'}}).$$

First, $T_{\Sigma/E} / \equiv_{E'} \ \cong \ T_{\Sigma, E \cup E'}$. Then, under the above assumptions, $\mathcal{R}/E' = (\Sigma, E \cup E', R)$ is k-deadlock free and

$$(\rightarrow^1_{\mathcal{R}/E'})^\bullet = \rightarrow^1_{\mathcal{R}/E'} = (\rightarrow^1_{\mathcal{R}})^\bullet / \equiv_{E'}.$$

Therefore, at a purely mathematical level, $\mathcal{R}/E'$ seems appropriate. Now, executability requires that the equations $E \cup E'$ are Church-Rosser and terminating, and that the rules R are coherent with respect to $E \cup E'$. To check or enforce these conditions, one can use the tools available in the Maude Formal Environment, as shown in Section 2.3.

Concerning the state properties in the quotient system, given its definition

$$L_{\Pi/\equiv_{E'}}([t]_{E\cup E'}) = \bigcap_{[x]_E \subseteq [t]_{E\cup E'}} L_\Pi([x]_E).$$

it may not be easy to come up with equations D' defining $L_{\Pi/\equiv_{E'}}$. But it becomes easy if the properties are preserved by E' in the following sense:

$$[x]_{E\cup E'} = [y]_{E\cup E'} \implies L_\Pi([x]_E) = L_\Pi([y]_E).$$

In this case we do not need to change the equations D and therefore we have

$$\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'} \cong \mathcal{K}(\mathcal{R}/E', k)_\Pi.$$

Property preservation can be proved inductively or, instead, one can use tools in the Maude Formal Environment to mechanically discharge the corresponding proof obligations.

Once E, E', and R satisfy all these executability requirements, by construction, the quotient simulation $\mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}, E)_\Pi/\equiv_{E'} \cong \mathcal{K}(\mathcal{R}/E', k)_\Pi$ is strict, so it reflects satisfaction of arbitrary LTL formulas. Moreover, since $\mathcal{R}/E'$ is executable, for an initial state t having a finite set of reachable states we can use the Maude model checker to check if a property holds. In this way, we model check on the abstract version the properties we are interested in checking for the original system.

4 Equational Abstraction on the Unordered-Channel Example

Let us go back to the unordered-channel example in Section 2.2. The rule

```
rl [snd]: { N ; L, M | C | P, J } => { N ; L, M | [N, M] C | P, J } .
```

allows sending several times the same message, but then the reachable state space is infinite. To identify repeated copies of sent messages, we add the following equation:

```
mod UNORDERED-CHANNEL-ABSTRACTION is
  including UNORDERED-CHANNEL .
  vars M N P K : Nats .    vars L L' : List .      var C : Conf .
  eq [A1]: { L, M | [N, P] [N, P] C | L', K }
         = { L, M | [N, P] C | L', K } .
endm
```

Verification of Executability Requirements. We can then check using the tools[3] in the Maude Formal Environment that the proposed abstraction is terminating, Church-Rosser, and sufficiently complete (although in the last case we get a warning due to the fact that the added equation is not linear, and therefore cannot be handled by the SCC tool).

```
Maude> (ct UNORDERED-CHANNEL-ABSTRACTION .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION is terminating.

Maude> (ccr UNORDERED-CHANNEL-ABSTRACTION .)
Maude> (submit .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION is Church-Rosser.

Maude> (scc UNORDERED-CHANNEL-ABSTRACTION .)
Warning: The functional module UNORDERED-CHANNEL-ABSTRACTION is
    sufficiently complete and has free constructors. However‘,
    module UNORDERED-CHANNEL-ABSTRACTION may still not be
    sufficiently complete or not have free constructors.
```

However, the coherence check fails because the ChC tool returns a critical pair:

```
Maude> (select tool ChC .)
The ChC has been set as current tool.

Maude> (cch UNORDERED-CHANNEL-ABSTRACTION .)
Coherence checking of UNORDERED-CHANNEL-ABSTRACTION
The following critical pairs cannot be rewritten:
  cp UNORDERED-CHANNEL-ABSTRACTION2 for A1 and rec
    { L:List,M:Nats | #3:Conf[N:Nats,J:Nats]| P:List,J:Nats }
    => { L:List,M:Nats | #3:Conf ack(J:Nats)[N:Nats,J:Nats]|
    P:List @ N:Nats ; nil,s(J:Nats) }.
   The sufficient-completeness, termination and Church-Rosser
    properties must still be checked.
```

In this particular example, the critical pair indicates that one can lose possible rewrites by applying first the equation and that this can be solved by adding the rule which provides the corresponding rewrite steps. Therefore, to recover coherence, we add the appropriate rule, which is just a simple renaming of the returned critical pair.

Since, after the equational abstraction, multiplicity of messages in the channel no longer matters, the new rule allows to receive a message without deleting it from the channel; thus, in the channel of the righthand side of the rule, we can see that the `[N, K]` message is kept in the channel together with the corresponding acknowledgement `ack(K)`.

[3] In the code shown in this section we omit some intermediate commands and show part of the output, to emphasize thus the final result.

```
mod UNORDERED-CHANNEL-ABSTRACTION-2 is
  including UNORDERED-CHANNEL-ABSTRACTION .
  vars M N K : Nats .    vars L L' : List .    var C : Conf .
  rl [snd2]: { L, M | [N, K] C | L', K }
          => { L, M | [N, K] ack(K) C | L' @ N ; nil, s(K) } .
endm
```

Now we can check that all the executability conditions are indeed satisfied; for instance, in checking coherence we get no critical pair this time.

```
Maude> (select tool ChC .)
The ChC has been set as current tool.

Maude> (cch UNORDERED-CHANNEL-ABSTRACTION-2 .)
Coherence checking of UNORDERED-CHANNEL-ABSTRACTION-2
    All critical pairs have been rewritten and no rewrite with rules can
    happen at non-overlapping positions of equations left-hand sides.
    The sufficient-completeness, termination and Church-Rosser properties
    must still be checked.
Maude> (submit .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-2 is coherent.
```

Verification of Property Preservation. We can now move to the specification of the properties. Here the essential property we are looking for is that the protocol achieves in-order communication in spite of the unordered channel. This property may be defined by means of a `prefix` property on lists, as done in the following module which imports a module `BOOLEAN` (not shown here) providing Boolean values and standard operations on them.

```
mod UNORDERED-CHANNEL-PROP is
  protecting BOOLEAN .
  protecting UNORDERED-CHANNEL .
  sort Prop .
  op _~_ : Nats Nats -> Bool .    *** equality predicate
  op _|=_ : State Prop -> Bool [frozen] .  *** satisfaction
  vars M N K P : Nats .     vars L L' L'' : List .    var  C : Conf .
  eq 0 ~ 0 = true .
  eq 0 ~ s(N) = false .
  eq s(N) ~ 0 = false .
  eq s(N) ~ s(M) = N ~ M .
  op prefix : List -> Prop [ctor] .
  eq [I1]:  { L', N | C | K ; L'', P } |= prefix(M ; L) =
     (M ~ K) and  { L', N | C | L'', P } |=  prefix(L) .
  eq [I3]: { L', N | C | nil, K } |=  prefix(L) = true .
  eq [I4]: {L', N | C | M ; L'', K } |=  prefix(nil) = false .
endm
```

We assume that all initial states are of the form

```
{n1 ; ... ; nk ; nil , 0 | null | nil , 0}
```

where the sender's buffer contains a list of numbers `n1 ; ... ; nk ; nil` and has its counter set to 0, the communication channel is empty, the receiver's buffer is also empty, and the receiver's counter is initially set to 0. The following module puts everything together and declares a concrete initial state.

```
mod UNORDERED-CHANNEL-ABSTRACTION-CHECK is
  extending UNORDERED-CHANNEL-ABSTRACTION-2 .
  including UNORDERED-CHANNEL-PROP .
  op init : -> State .
  eq init = {0 ; s(0) ; s(s(0)) ; nil , 0 | null | nil , 0} .
endm
```

It is easy to see that the set of abstract states is finite and that the module `UNORDERED-CHANNEL` is deadlock free. Moreover, to show property preservation, we can check that the equations in both modules `UNORDERED-CHANNEL-PROP` and `UNORDERED-CHANNEL-ABSTRACTION-CHECK` are terminating, Church-Rosser, and sufficiently complete, and rules are still coherent.

```
Maude> (ct UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-CHECK is terminating.

Maude> (ccr UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Maude> (submit .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-CHECK is Church-Rosser.

Maude> (scc UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Maude> (submit .)
Warning: The functional module UNORDERED-CHANNEL-ABSTRACTION-CHECK
   is sufficiently complete and has free constructors. However [...]

Maude> (cch UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Maude> (submit .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-CHECK is coherent.
```

Model Checking the Property. Finally, we can model check the desired property on the abstract version of the unordered communication channel, as follows:

```
mod UNORDERED-CHANNEL-ABSTRACTION-MODEL-CHECK is
   including UNORDERED-CHANNEL-ABSTRACTION-CHECK .
   including LTL-SIMPLIFIER .   *** optional
   including MODEL-CHECKER .
endm

Maude> reduce in UNORDERED-CHANNEL-ABSTRACTION-MODEL-CHECK :
     modelCheck(init, []prefix(0 ; s(0) ; s(s(0)) ; nil)) .
rewrites: 361 in 41ms cpu (42ms real) (8780 rewrites/second)
result Bool: true
```

The property then holds also in the original system, as justified in Section 3.

5 Concluding Remarks

The equational abstraction technique introduced in [5] and summarized here is fairly simple and takes advantage of the expressiveness of rewriting logic and its Maude implementation [1], as well as of the tools available in the Maude Formal Environment [3]. Other examples are available in the references, but they do not use the Maude Formal Environment in its current integrated form, as we have done with the main example in this paper.

Related work includes the generalization of the equational theory extension $(\Sigma, E) \subseteq (\Sigma, E \cup E')$ to theory interpretations $(\Sigma, E) \longrightarrow (\Sigma', E'')$ and also to (stuttering) simulations, studied in detail in [6].

Future work will be dedicated to improving the interface of the Maude Formal Environment to make it more user-friendly. Also, the Inductive Theorem Prover (ITP) needs more and better integration with the other tools.

Acknowledgments. We are very grateful to our colleagues José Meseguer and Miguel Palomino, whose work on equational abstraction is summarized in this paper; the organizers of *CBSoft* and *SBMF 2014* for their invitation to present this work in such a nice environment; and Christiano Braga for all his enthusiastic help.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
2. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. Journal of Logic and Algebraic Programming **81**(7–8), 816–850 (2012)
3. Durán, F., Rocha, C., Álvarez, J.M.: Towards a Maude Formal Environment. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 329–351. Springer, Heidelberg (2011)
4. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 184–191. Springer, Heidelberg (2014)
5. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science **403**(2–3), 239–264 (2008)
6. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. Journal of Logic and Algebraic Programming **79**(2), 103–143 (2010)

http://www.springer.com/978-3-319-15074-1

Formal Methods: Foundations and Applications
17th Brazilian Symposium, SBMF 2014, Maceió, AL, Brazil,
September 29--October 1, 2014. Proceedings
Braga, C.; Martí-Oliet, N. (Eds.)
2015, IX, 179 p. 39 illus., Softcover
ISBN: 978-3-319-15074-1