Model Checking TLR* Guarantee Formulas on Infinite Systems

Autor: Óscar Martín
Directores: Alberto Verdejo y Narciso Martí-Oliet

Trabajo de fin de Máster en Programación y Tecnología del Software
Curso 2012–2013

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Calificación: sobresaliente

Autorización de difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo de Fin de Máster: "Model Checking TLR* Guarantee Formulas on Infinite Systems", realizado durante el curso académico 2012-2013 bajo la dirección de Alberto Verdejo y Narciso Martí en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Óscar Martín
Julio de 2013

Resumen

Presentamos la implementación de un *model checker* para sistemas con una cantidad de estados potencialmente infinita. Se ha desarrollado sobre el lenguaje y el sistema Maude, basado en lógica de reescritura. Los sistemas que se quieran analizar también han de estar especificados como módulos Maude. El *model checker* funciona con estados explícitos. Así, en sistemas infinitos, no podemos esperar que la comprobación acabe en todos los casos. De hecho, solo proporciona un semi-algoritmo para validar fórmulas de garantía (o, equivalentemente, para invalidar fórmulas de seguridad). Para evitar entrar en caminos infinitos, las búsquedas siempre se llevan a cabo con profundidad acotada.

La lógica temporal que usamos es TLR* (*Temporal Logic of Rewriting*). Esta lógica es una generalización de CTL* que usa proposiciones atómicas no solo sobre estados, sino también sobre transiciones, proporcionando así una mayor potencia expresiva. Como paso intermedio, presentamos un lenguaje de estrategias para Maude. Las fórmulas de garantía se traducen primero a expresiones de estrategia y, entonces, se hace *evolucionar* en paralelo al sistema y a la estrategia para buscar cómputos que satisfagan la estrategia y, por tanto, la fórmula.

Se incluyen varios ejemplos para mostrar la utilidad de nuestra herramienta. En particular, se presenta un ejemplo más largo, relativo a protocolos de coherencia de caché.

Las tres ideas en las que se basa este trabajo —el *model checker*, TLR* y el lenguaje de estrategias— son propuestas tomadas de [21].

Palabras clave

Sistema de infinitos estados, lógica de reescritura, Maude, *model checking*, estrategia, lógica temporal, TLR*, fórmula de garantía, protocolo de coherencia de caché.

Abstract

We present the implementation of a model checker for systems with a potentially infinite number of states. It has been developed in the rewriting-logic language and system Maude. The systems to be analysed need also be specified as Maude modules. The model checker is explicit-state, that is, not symbolic. Thus, in infinite systems, we cannot expect it to finish in every case. Indeed, it only provides a semi-decision algorithm to validate guarantee formulas (or, equivalently, to falsify safety ones). To avoid getting lost in infinite paths, search is always done within bounded depth.

The temporal logic to which the formulas belong is TLR*, the Temporal Logic of Rewriting. This is a generalization of CTL* that uses not only atomic propositions on states but also on transitions, providing, in this way, a richer expressive power. As an intermediate step, a strategy language for Maude is presented. The guarantee formulas are first translated into strategy expressions and, then, the system and the strategy *evolve* in parallel searching for computations that satisfy the strategy and the formula.

A number of examples are included, showing the usefulness of the tool. In particular, a longer example on cache coherence protocols is presented.

The three ideas on which this work is based, that is, the model checker, TLR*, and the strategy language, are proposals taken from [21].

Keywords

Infinite-state system, rewriting logic, Maude, model checking, strategy, temporal logic, TLR*, guarantee formula, cache coherence protocol.

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 A Simple Example . . . 3
2.2 A Quick Maude Primer . . . 4
2.3 Rewriting Logic . . . 6
2.3.1 Rewriting Theories . . . 7
2.3.2 Proof Terms and Computations . . . 8
2.4 Temporal Logics and TLR* . . . 10
2.4.1 State-Based and Action-Based Temporal Logics . . . 10
2.4.2 Spatial Actions . . . 11
2.4.3 Syntax and Semantics of TLR* . . . 13
2.4.4 Sublogics of TLR* . . . 15
2.5 A Strategy Language . . . 15
2.5.1 Strategies . . . 15
2.5.2 A Strategy Language . . . 16
2.5.3 Strategy Semantics . . . 17
2.5.4 Guarantee Formulas as Strategies . . . 19
2.6 Model Checking for Finite and Infinite Systems . . . 20

3 The Model Checker **21**

3.1 Our TLR* Flavor . . . 21
3.1.1 Transition Proof Terms . . . 21
3.1.2 Transition Patterns . . . 22
3.1.3 Atomic Propositions on Transitions . . . 24
3.2 Strategy Semantics . . . 24
3.3 The Main Loop . . . 34
3.4 Some Notes on the Implementation . . . 36
3.4.1 `EXTENDED` . . . 36
3.4.2 Performance Improvements . . . 38
3.5 A Brief Manual . . . 38
3.5.1 The Commands . . . 38
3.5.2 The Modules . . . 40

4 Examples **41**

4.1 Faulty Channels, Attackers, and Cookies . . . 41
4.2 Faulty Channels and Timing . . . 45

4.3 Production Rules for Grammars . . . 48
4.4 A Finite System: MUTEX . . . 49

5 A Case Study: The MSI Cache Coherence Protocol **52**
5.1 The Setting . . . 52
5.2 The MSI Protocol . . . 54
5.3 Level 1 Model . . . 56
5.4 Level 2 Model . . . 63
5.5 Level 3 Model . . . 66

6 Related Work **69**
6.1 Model Checking TLR* . . . 69
6.2 Infinite Systems . . . 70
6.3 Model Checking Cache Coherence Protocols . . . 70
6.4 Strategies . . . 71

7 Conclusions and Future Work **72**

Bibliography **74**

Chapter 1

Introduction

Rewriting logic as a formalism for system specification began in the early 1990's with several papers by J. Meseguer [20]. In addition to being a specification language, it is an executable logic, which makes it a very useful formalism for system modeling. Maude is a rewriting logic-based language and developing system that incorporates both equational logic and rewriting logic [10]. Parallelism and nondeterminism are natural features of rewriting logic and Maude.

One of the main goals of formal system specification is to formally reason on the given systems. Model checking [9], that is, the automatic and exhaustive verification of properties of systems, is therefore a natural jobmate for Maude. Indeed, Maude includes a model checker for temporal formulas on the logic LTL.

This master thesis presents a model checker, based on Meseguer's proposals in [21], that has two main novelties: it works on systems with a potentially infinite number of states, and their temporal properties are formulas of (a subset of) the logic TLR*. On the first point, it must be noted that several studies and methodologies exist for dealing with infinite systems ([1, 16, 22], among many others). However, most of them work with some kind of abstraction or similar mechanism that turns them finite in essence or, at least, finitely representable. Of course, dealing with infinite systems as such there is the risk of nontermination. Indeed, the model checker we present here provides only a semi-decision algorithm to verify TLR* guarantee formulas or, equivalently, to falsify safety formulas.

TLR* is a temporal logic proposed as well in [21] as a companion for rewriting logic. Most temporal logics are either state-based or action-based, in the sense that their atomic propositions are properties either of states or of transitions, but not both. TLR* is a generalization of CTL* that includes the capability of talking about both kinds of properties in the same formula. As Meseguer showed, some properties of systems can be naturally expressed only within such a framework.

Strategies [11, 24, 7, 19], applied to system specifications, are a means of guiding their evolution and restricting their nondeterminism. There exists a rich strategy language for Maude. Once again following [21], we have implemented another strategy language. Model checking is performed by first translating the given TLR* guarantee formula into a strategy expression and, then, checking whether the system can evolve according to the resulting

strategy.

More concretely, we are model checking *strategy formulas*, that is, strategy expressions with a leading universal or existential path-quantifier. A universal (resp. existential) strategy formula represents the question: Does *every* (resp. *some*) possible evolution of the system satisfies the given strategy expression? Correspondingly, the temporal logic that we are able to model check is the subset of TLR* of path-quantifier-free, linear-time formulas with just a leading path quantifier. These are the TLR* formulas we are able to translate into strategy formulas.

Often, when using strategies, one is interested in the whole set of paths in a system that satisfy the given strategy —or maybe just the states to which those paths lead to. For our purposes, we do not need that whole set, but just need to know that *some* path does satisfy the strategy —to get a positive answer to an existential question— or that *some* path does not —for a negative answer to a universal question. That has guided our choice of implementation, which consists in a recursive bounded backtracking. Each step in that backtracking is given by the evolution of both the system and the strategy, so that the child (state, strategy) pair is a condition for the parent pair to hold.

Bounded search is necessary to avoid getting lost into an infinite branch when, perhaps, the answer (or a best or shorter answer) is on another branch. Our implementation provides a way to specify the maximum depth to be explored. Also, it provides a command `deeper` to ask the system to explore some more levels based on the open branches left by a previous model-checking command.

Before delving deeper into the workings of our model checker, all the concepts mentioned above deserve explanation or, at least, informal introduction. We do this in the next chapter with the help of a simple example. Then, the concepts underlying the model checker are presented and some details of the implementation are visited. A collection of examples follows, showing the different aspects and uses of our tool. We devote a complete chapter to a longer case study on the MSI cache coherence protocol. We finish with some related works and the conclusions.

The complete Maude specifications for the model checker as well as for the examples that are described in this work are available at http://maude.sip.ucm.es/ismc.

Chapter 2

Preliminaries

In this chapter we introduce all the subjects that have a role in the construction of our model checker. The material included is not new, though the presentation is original in many points, as is the example we use.

2.1 A Simple Example

Consider this situation: A series of electronic devices are set to count the number of people that come into a room through all of its doors. Each device is watching a door and counting. So that all devices have available the same, global information on the total of people, they exchange their data: each time a person enters through one door, the device watching that door sends a message to its fellows informing them of the fact. This can be, if you wish, a metaphor behind the system we now describe in a less fancy manner.

There are a number of devices. In order to be able to share data, they are organized as a ring, so that each device knows to whom it must send its messages, that are resent until they have visited the whole ring. As we do not care about the sources of data, in our model each device is able by itself to increase its counter by one. Each time this happens, it must send a message telling the news to its *next* device.

Figure 2.1 shows a ring composed by three devices. Some "+1" messages are shown too. A message has just left device A and is waiting to be processed by device B. Another "+1" message that was sent by device A is about to complete the whole ring and be removed.

Thus, we identify three actions in the system:

change, by which a device simulates counting one more person and informs about it;

resend, by which a device receives a message, updates its counter according to it, and resends the same information to another (the next) device;

remove, by which a device receives a message sent by itself and removes it.

This is the example we use in what follows.

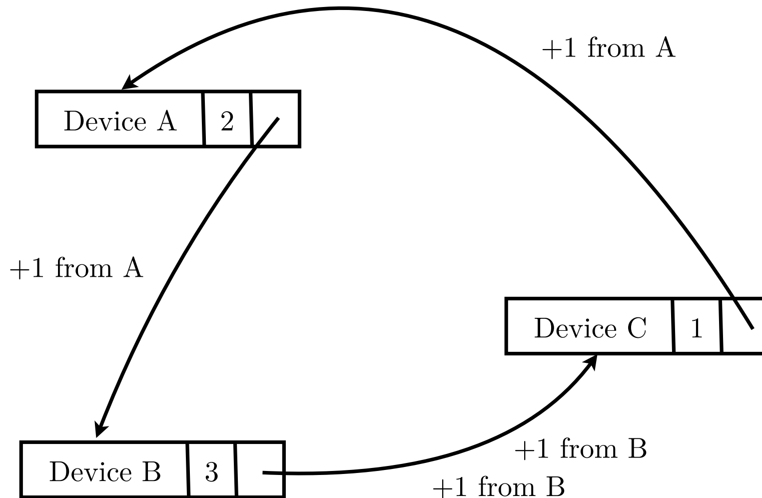

Figure 2.1: A ring with three devices

2.2 A Quick Maude Primer

We are going to review a complete Maude specification for the example system. It is divided into pieces with comments in between, whose aim is to serve as an informal and quick introduction to Maude. See [10] for all the details on Maude.

All system modules in Maude have this shape:

```
mod COUNTING is
    system specs go here
endm
```

In this case, we have given our module the name `COUNTING`. First, an import:

```
protecting NAT .
```

It is telling the Maude system that we are using the natural numbers, that are implemented in module `NAT`. Next, we declare the types of objects the system is using:

```
sort Id .
subsort Nat < Id .
sort Device .
op [_,_,_] : Id Id Nat -> Device [ctor] .
```

This specifies two types `Id` and `Device` (they are called `sorts` in Maude's terminology). Maude's type system is order-sorted: the `subsort` statement says that any `Nat` is also an `Id`. As we are not giving any other way to build `Id`s, both are synonyms, and the new name just tries to add clarity. The way to build a `Device` is to enclose between square brackets two `Id`s and a `Nat`: `[1, 2, 17]`. The idea is that the first component is the device's `Id`, the second is the `Id` of the next device in the ring, and the third is the amount this device knows has been counted up to this moment. We need more sorts:

```
sort Message .
op _|>_ : Id Id -> Message [ctor] .
```

These two lines define what a `Message` is. As the direction of the arrow implies, the first argument is the sender's `Id`, and the second the addressee's `Id`. The last sort we need contains the state of the system:

```
sort State .
subsorts Device Message < State .
op nullState : -> State [ctor] .
op __ : State State -> State [ctor comm assoc id: nullState] .
```

We declare that any `Device` or `Message`, by itself, constitutes a `State`. We also provide an operator with empty syntax `__` that allows to juxtapose any number of `State`s to get a new one. Note the `comm` and `assoc` attributes given to the operator. This way of defining states is an usual idiom in Maude. Also note that we have declared a `nullState` to be used as identity element for states.

Now, a few variables to be used later:

```
vars I J N : Id .
var A : Nat .
```

Finally, we get to the rules for the dynamic behavior of the system. Each rule has a label (between square brackets), a left-hand-side term, the symbol `=>`, and a right-hand-side term. Some rules include conditions. Our three rules rewrite terms of sort `State`, but any other sort declared in the module is valid in a rewriting rule.

One important point is that it is not the whole state of the system that has to match a left-hand-side term: whenever a subterm matches the left-hand side, the whole term can be rewritten in the expected way. These are the rules:

```
rl [change] : [I, N, A] => [I, N, s(A)] (I |> N)
crl [resend] : [I, N, A] (J |> I) => [I, N, s(A)] (J |> N)
   if I =/= J .
rl [remove] : (I |> I) => nullState .
```

The rules should be easy to understand. The function `s` is the successor, that is defined in module `NAT`. Rule `resend` states that when device `I` sees a message addressed to itself, it updates its data and resends the same data to the next device. Note that the sender is still the one who originated the message. It is a conditional rule, because this should only happen for devices other than the original one. Rule `remove` just drops a message whose sender and addressee coincide —that is, the message has already visited the whole ring. Note the use in this rule of the variable `I` twice: this rule should only be applied when both components of a message are equal.

For convenience, we code a function that creates rings of any given size, to be used as initial states for the system. So we add these final lines to `COUNTING`, that are not properly part of the system's specification:

```
vars X X' : Nat .
```

```
op ring : Nat -> State .
eq ring(0) = nullState .
eq ring(s(X)) = ring(0, X) .
op ring : Nat Nat -> State .
eq ring(X, X) = [X, 0, 0] .
eq ring(X, X') = [X, s(X), 0] ring(s(X), X') [owise] .
```

There are two functions called `ring` that are different because of the number of arguments they take. The coding of functions follows a functional style. Pattern matching on the shape of the arguments is used when there are several equations for the same operator. The `owise` (otherwise) attribute tells that equation can only be used when no other is enabled, that is, in our case, when the two arguments are different.

Let us begin using Maude commands: This one tests the function `ring`:

```
reduce ring(4) .
```

Maude answers the result of reducing `ring(4)` to its normal form and its sort:

```
result State: [0,1,0] [1,2,0] [2,3,0] [3,0,0]
```

More interesting is asking Maude for some rewrites from a given state. This asks for three successive rewrites:

```
rewrite [3] ring(4) .
```

This is Maude's answer:

```
result State: (0 |> 3) [0,1,1] [1,2,1] [2,3,1] [3,0,0]
```

In general, several rules can be applied to a given state. The `rewrite` command chooses just one of the possible behaviors of the system. Another interesting command is `search`, that explores all behaviors:

```
search [,5] ring(4) =>* S:State (7 |> 7) .
```

Here we are asking Maude to perform five consecutive rewritings in all possible ways to try to arrive to a state with a message from device `7` to itself. No rule in the system allows for the dynamic creation of devices, and device `7` does not exist at start, so Maude readily informs us that no solution has been found.

2.3 Rewriting Logic

A note to the reader: In this and the next section, we describe rewriting theories, TLR* and its associated concepts in the way Meseguer introduced them in [21]. However, the way we use them for our implementation is not exactly like this. The slight variations needed are described later, in Chapter 3. Thus, some concepts are introduced twice with not identical descriptions. This may seem somewhat messy, but we wanted to keep a clear separation between preexisting theory and the work that, based on it, has led to the implementation of our tool.

2.3.1 Rewriting Theories

Formally, a rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, where Σ is a many-sorted signature, E a set of equations and R a set of rewriting rules of the form $l : q \to q'$, with l a label, and q, q' terms of the same sort and such that all variables in q' appear also in q. Such a triple specifies a concurrent, nondeterministic system in the following way: The states of the system are E-equivalence classes of ground terms $[t]_E$; told another way, the initial algebra $T_{\Sigma/E}$ constitutes the state space. The dynamics of the system are given by the rewrite rules in R. As states are equivalence classes of terms, rewriting must be understood also at this level. That is, a transition from state $[t]_E$ to state $[t']_E$, denoted by $[t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$, is possible in $\mathcal{R}$ iff there exists $u \in [t]_E$ and $u' \in [t']_E$ such that u can be rewritten to u' using some rule $l : q \to q'$ in R.

The rewriting of term u by rule l is possible whenever a subterm of u matches q. To explain this in more detail, we need some notation first: $u|_p$ is the subterm of u at position p, and $u[s]_p$ is the term that results from u by removing its subterm at position p and inserting s in its place. (Positions are a precise way to identify subterms of a term based on its syntactic tree. See, for instance, [2] for details.) Now, say $u_0 = u|_p = \theta(q)$ is the matching subterm of u. Then, let us call $u'_0 = \theta(q')$, for the same θ that produced u_0. Then, the result of rewriting u by l, is the term $u' = u[u'_0]_p$, that is, the term u with u_0 removed and u'_0 inserted in its place.

For arbitrary E and R, whether $[t]_E \longrightarrow^*_{\mathcal{R}} [t']_E$ holds is undecidable in general. That is why the most useful rewrite theories satisfy additional executability conditions. A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ (where the set of equations has been split into two disjoint subsets) is *computable* if E, A and R are finite and the following conditions hold:

1. Equality modulo A is decidable, and there exists a matching algorithm modulo A, producing a finite number of A-matching substitutions or failing otherwise, that can implement rewriting in A-equivalence classes. This implies that for a rewrite theory of the form $\mathcal{R}' = (\Sigma, A, Q)$ with Q a finite set of rewrite rules, it is decidable whether $[t]_A \longrightarrow^1_{\mathcal{R}'} [t']_A$ holds or not.

2. $(\Sigma, E \cup A)$ is ground terminating and confluent modulo A. That is: (i) there are no infinite sequences of rewritings with E modulo A; and (ii) for each $[t]_A \in T_{\Sigma/A}$ there is a unique A-equivalence class $[\mathrm{can}_{E/A}(t)]_A \in T_{\Sigma/A}$, called the *$E$-canonical form of* $[t]_A$ *modulo* A, such that the last term, which cannot be further rewritten with E modulo A, of any terminating sequence beginning at $[t]_A$ is necessarily $[\mathrm{can}_{E/A}(t)]_A$.

3. The rules R are ground coherent relative to the equations E modulo A. That is, if $[t]_A$ is rewritten to $[t']_A$ by a rule $l \in R$, then $[\mathrm{can}_{E/A}(t)]_A$ is also rewritten by the same rule l to some $[t'']_A$ such that $[\mathrm{can}_{E/A}(t')]_A = [\mathrm{can}_{E/A}(t'')]_A$.

These three conditions imply that for each sort $s \in \Sigma$ the relation $\longrightarrow^1_{\mathcal{R},s}$ is a computable binary relation on $T_{\Sigma/E\cup A,s}$: one can decide $[t]_{E\cup A} \longrightarrow^1_{\mathcal{R}} [t']_{E\cup A}$ by generating the finite set of all one-step R-rewrites modulo A of $\mathrm{can}_{E/A}(t)$ and testing if any of them has the same E-canonical form modulo A as $[\mathrm{can}_{E/A}(t')]_A$.

The three conditions are quite natural and are typically met in practical rewriting logic specifications. In Maude, the set of equations A is given by *operator attributes* like `comm` and `assoc` used in the example in Section 2.2, for which Maude knows specific matching algorithms. Also, Maude's equations are considered left to right —as they are in functional programming— and Maude does not try to apply a rewriting rule until it has reduced the term to its E-irreducible form modulo A, that is, to its canonical form. By the way, Maude also allows for other features like an order-sorted type system, conditional equations and rules, and membership equational logic. We skip these and other features in this introduction.

In addition to being computable, we assume that the systems we work with satisfy also some technical properties that make our lives easier:

- One of the sorts in their signature must be called `State`. As one can expect, it corresponds to the sort of the top-level terms we want our system to work on.

- If $\mathcal{R}$ has a sort named `StateProp`, then it must also have a sort named `Bool` with constants `true` and `false` and an operator `|=` : `State` $\times$ `StateProp` $\rightarrow$ `Bool`. The idea is that `StateProp` is the designated sort of atomic state propositions, and `|=` is the relation defining whether a given state satisfies a given state proposition. Furthermore, if Σ is the signature of $\mathcal{R}$, then we define the subsignature $\Sigma_S \subseteq \Sigma$ of its state-proposition symbols as the set of all operators in Σ whose result sort is `StateProp`. Also, we call Π_S the set of terms of sort `StateProp`.

- They must be deadlock-free, that is, no sequence of rewritings can lead us to a term that cannot be further rewritten. This is not such a strong condition as it can seem since any theory (with the mild proviso of not having rules with rewrites in their conditions) with deadlocks can be transformed into a deadlock-free semantically-equivalent one. The trick is basically to add a self loop to any deadlock state. See [10], for instance. Sometimes, the actual Maude modules we use in our examples have deadlocks. The point is that the theoretical exposition is valid for all of them, through the self-loop trick.

2.3.2 Proof Terms and Computations

Terms of sort `State` represent states of the system. The transitions from one state to another can be also represented by terms (on a larger signature) that are called *proof terms*. In this way, both states and transitions are first-class citizens, with their structures and their properties.

The inference rules of rewriting logic [20, 8] allow to infer all concurrent computations possible in the system specified by $\mathcal{R}$. That is, given two states $[u], [u'] \in T_{\Sigma/E \cup A}$, one can reach $[u']$ from $[u]$ by some possibly complex concurrent computation if and only if one can prove $\mathcal{R} \vdash [u] \rightarrow^{+} [u']$. In rewriting logic any such complex computation reaching $[u']$ from $[u]$ is witnessed by a *proof term*. General proof terms can include sequential as well as concurrent one-step proof terms. However, proof terms are identified modulo some natural

equations, substituting concurrency by interleaving, making any proof term equivalent to a sequential composition of one-step proof terms.

One-step proof terms can be characterized in an algebraic fashion. Given a rewrite $[u] \to_{\mathcal{R}} [u']$ using a rule $l : q \to q'$, its proof term has the form $v[l(u_1, \ldots, u_n)]_p$, where $v \in [u]$, p is the position in v where the rule is applied, and $u_1, \ldots, u_n$ are Σ-terms. In more practical words, the one-step proof term $v[l(u_1, \ldots, u_n)]_p$ indicates that:

- the rule with label l has been used;
- the substitution $x_1 \mapsto u_1; \ldots; x_n \mapsto u_n$ has been used to match q with a subterm of v, where $x_1, \ldots, x_n$ are all the variables in q in the textual order in which they appear;
- the context in which the rewriting happened is $v[\ ]_p$.

Let us illustrate this with the `COUNTING` example. Say the system is in state

```
[1, 0, 5] [0, 1, 5]
```

and, by applying the rule

```
rl [change] : [I, N, A] => [I, N, s(A)] (I |> N) .
```

we arrive to

```
[1, 0, 5] [0, 1, 6] 0 |> 1
```

Then, the one-step proof term that witnesses the rewriting is given by

- the label `change`;
- the substitution `I` $\mapsto$ `0`; `N` $\mapsto$ `1`; `A` $\mapsto$ `5`;
- the context `[1, 0, 5] []`, where the symbol `[]` marks the place where the rewriting took place.

In algebraic fashion, this is the proof term: `[1, 0, 5] change(0, 1, 5)`. (Remember there is an implicit, empty-syntax operator joining the elements in a `State`, and also the two elements in this proof term.)

One-step proof terms are useful to represent atomic transitions in computations, in the same way as terms of sort `State` represent states. We just need a few more technical details to make the treatment of such one-step proof terms *canonical*. The point is that different terms in $[t]_{E \cup A}$ could be rewritten with different instantiations of the same rule. In our example, the context

```
[1, 0 + 0, 7] []
```

is also valid in the previous proof term, as our module knows about natural numbers. A straightforward notion of canonicity can be defined (we skip the definition) on contexts and on the terms to be used in substitutions that leaves us with the desired notion of *canonical one-step proof term*, whose set we denote as $\mathrm{CanPTerms}^1(\mathcal{R})$.

One more bit of notation: $(\mathrm{Can}_{\Sigma/E\cup A})_K$ is the set of all A-equivalence classes of the form $[\mathrm{can}_{E/A}(u)]_A$, where u is a ground-term of sort K. In particular, $(\mathrm{Can}_{\Sigma/E\cup A})_{\texttt{State}}$ describes the set of all states of the system specified by $\mathcal{R}$ in their canonical form representation.

Finally, we can state the following definition:

Definition 2.1. *An* infinite computation *in $\mathcal{R}$ is a pair of functions*

$$s : \mathbb{N} \to (\mathrm{Can}_{\Sigma/E\cup A})_{\texttt{State}} \qquad \textit{and} \qquad t : \mathbb{N} \to \mathrm{CanPTerms}^1(\mathcal{R})$$

such that for all $n \in \mathbb{N}$ we have $s(n) \xrightarrow{t(n)} s(n+1)$. When convenient, we write $s_i = s(i)$ and $t_i = t(i)$. That is,

$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots$$

A finite computation *is a prefix of an infinite one, with the understanding that it contains one more state than transitions:*

$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_{n+1}$$

When we describe TLR*, we will see that computations are the semantic entities on which the truth of their formulas is evaluated.

2.4 Temporal Logics and TLR*

The Temporal Logic of Rewriting, TLR*, was designed to get the best of both worlds: the state-based logics and the action-based logics. We describe its syntax and semantics in this section, with the previous necessary definition of *spatial action*.

2.4.1 State-Based and Action-Based Temporal Logics

Temporal logics, in their different flavors, are the usual formalism to express the properties we expect a system to satisfy as it evolves in the future. In addition to the classical logical operators, a temporal formula can include temporal operators. In this work we use the operators **A**, **E**, **G**, **F**, **X**, **U**, **R**, and **W**, with the following intuitive meaning (see later for formal definitions):

$\mathbf{A}\,\phi$: every possible computation from the current state will satisfy ϕ;

$\mathbf{E}\,\phi$: some possible computation from the current state will satisfy ϕ;

$\mathbf{G}\,\varphi$: at every future time in the computation φ will hold;

$\mathbf{F}\,\varphi$: at some future time in the computation φ will hold;

$\mathbf{X}\,\varphi$: at the next point in time φ will hold;

$\varphi_1\,\mathbf{U}\,\varphi_2$: at some future time φ_2 will hold and, until then, φ_1 will always hold;

$\varphi_1 \, \mathbf{R} \, \varphi_2$: φ_2 is true until the future time in which φ_1 is true, if this ever happens;

$\varphi_1 \, \mathbf{W} \, \varphi_2$: φ_1 will hold at every future time until φ_2 does, if this ever happens;

As implied by the notation used above, there are two classes of temporal formulas: on states, like $\mathbf{A}\,\phi$, and on computations, like $\mathbf{G}\,\varphi$. More on this later.

Usually, temporal logics fall in one of two classes according to the kind of atomic propositions they use: state-based or action-based. State-based logics, like LTL, CTL and CTL*, can only talk directly about states. Action-based logics, like A-CTL and Hennessy-Milner logic, can only talk directly about actions (that is, transitions).

As Meseguer shows in [21], some properties of systems can be naturally expressed only using atomic propositions of both classes. Consider again our example system `COUNTING`. To avoid cluttering in our system's communication channel, we would like that *self messages* like `I |> I` are removed as soon as they appear, that is, we would like the formula

$$\text{there-is-self-msg} \rightarrow \text{remove-is-executed}$$

always to hold, where the literals are atomic propositions, the first on states, the second on transitions. System `COUNTING`, by the way, as it stands, is not ready to always satisfy this formula.

A good source of examples for formulas using both kinds of atomic propositions are fairness conditions. For instance, we would like that each rule that is available infinitely often is run infinitely often. Fairness for a rule l can be formalized as

$$\mathbf{G}\,\mathbf{F}\,\text{enabled}(l) \rightarrow \mathbf{G}\,\mathbf{F}\,\text{taken}(l)$$

Here, enabled(l) is a proposition on states and taken(l) on transitions.

Some formalisms for system specification focus on states and form a good tandem with state-based temporal logics. Others are transition-oriented. Rewriting logic provides algebraic structure both to states (through terms of type `State`) and to actions (through one-step proof terms). The logic TLR* was designed by Meseguer in [21] to form a good tandem with rewriting logic.

In TLR*, there is, as usual, a signature, that we have called Σ_S, of atomic state propositions. But there are also atomic transition propositions given by patterns, the so called *spatial actions*, that we explain next.

2.4.2 Spatial Actions

Spatial actions are patterns for one-step proof terms, that is, patterns for transitions. Being patterns, they can (and will) be used as propositions on transitions: a transition satisfies the proposition (represented by the pattern) iff it matches the pattern. Spatial actions are terms on a signature that we denote as $\Omega(L)$, where L is the set of labels of rules in R. Let us formally define this signature.

In practice, typical system specifications have a clear division between *constructors* and other function symbols, so that a term is irreducible iff it is built only on constructors. For

instance, in the natural numbers, one surely expects that any term containing a '+' operator can be reduced, but terms containing just the constant '0' and the successor function cannot. In the system `COUNTING`, the function `ring` is not a constructor, as any term containing it can be reduced to some other term not containing it. Constructors can be marked in Maude, by the way, by the presence of the attribute `ctor`. We will assume our rewriting systems to have this separation between constructor symbols and other function symbols. Then, let Ω be the subsignature of constructors in Σ. Making Ω explicit and the assumption above are necessary so to make decidable the "instance of" relation between a one-step proof term and a spatial action pattern.

Now, the signature $\Omega(L)$ extends Ω adding the following:

- a fresh sort Top;
- for each rewrite rule $l : q \to q'$ in R with q, q' of sort S, and with the (textually ordered) variables $x_1, \dots, x_n$ in q having sorts $S_1, \dots, S_n$:
  - a constant l of sort S,
  - an operator $l : S_1 \times \cdots \times S_n \to S$,
  - an operator $\text{top} : S \to \text{Top}$.

On this signature, we define $\mathcal{R}$'s spatial action patterns, $\text{SP}(\Omega, L)$. They are a subset of $T_{\Omega(L)/A}(X)$, where X is a set of variables, with enough variables of each sort in Ω. The set $\text{SP}(\Omega, L)$ contains, for each $l \in L$:

- $[l]_A$;
- $[\text{top}(l)]_A$;
- $[l(u_1, \dots, u_n)]_A$, with $[l(u_1, \dots, u_n)]_A \in T_{\Omega(L)/A}(X)$ and $u_1, \dots, u_n \in T_{\Omega/A}(X)$;
- $[\text{top}(l(u_1, \dots, u_n))]_A$, with $[l(u_1, \dots, u_n)]_A \in \text{SP}(\Omega, L)$;
- $[v[l]_p]_A$, with p not the empty (top) position, $v \in T_{\Omega/A}(X)$, and $[v[l]_p]_A \in T_{\Omega(L)/A}(X)$;
- $[v[l(u_1, \dots, u_n)]_p]_A$, with p not the empty (top) position, $v, u_1, \dots, u_n \in T_{\Omega/A}(X)$, and $[v[l(u_1, \dots, u_n)]_p]_A \in T_{\Omega(L)/A}(X)$.

In our implementation, as explained later, we specify these patterns in a similar way to proof terms: as triples (context, rule label, substitution) —though not the three elements need to be present in a pattern— and with the indication top where needed.

Finally, we need to make precise when a proof term matches a pattern, that is denoted as $\gamma \sqsubseteq_A \delta$, for γ a proof term and $\delta \in \text{SP}(\Sigma, L)$. Let $[u]_A \preceq_A [u']_A$ iff there is a many-sorted substitution θ such that $[u]_A = [\theta(u')]_A$. This is a decidable relation by our assumption that there is an A-matching algorithm and the fact that they are only built on the constructors in Ω: it amounts to checking whether u matches the pattern u' modulo A. Now:

- $[v[l(u_1,\ldots,u_n)]_p]_A \sqsubseteq_A [l]_A$;
- $[l(u_1,\ldots,u_n)]_A \sqsubseteq_A [\mathrm{top}(l)]_A$;
- $[v[l(u_1,\ldots,u_n)]_p]_A \sqsubseteq_A [l(u'_1,\ldots,u'_n)]_A$ iff $[l(u_1,\ldots,u_n)]_A \preceq_A [l(u'_1,\ldots,u'_n)]_A$;
- $[l(u_1,\ldots,u_n)]_A \sqsubseteq_A [\mathrm{top}(l(u'_1,\ldots,u'_n))]_A$ iff $[l(u_1,\ldots,u_n)]_A \preceq_A [l(u'_1,\ldots,u'_n)]_A$.
- $[v[l(u_1,\ldots,u_n)]_p]_A \sqsubseteq_A [v'[l]_{p'}]_A$ iff $[v[l]_p]_A = [v'[l]_{p'}]_A$;
- $[v[l(u_1,\ldots,u_n)]_p]_A \sqsubseteq_A [v'[l(u'_1,\ldots,u'_n)]_{p'}]_A$
  iff $[v[l(u_1,\ldots,u_n)]_p]_A \preceq_A [v'[l(u'_1,\ldots,u'_n)]_{p'}]_A$;

Some examples, one for each case:

- `[1, 0, 7]` `change`$(0,1,5) \sqsubseteq_A$ `change`;
- `[1, 0, 7]` `change`$(0,1,5) \not\sqsubseteq_A \mathrm{top}($`change`$)$;
- `[1, 0, 7]` `change`$(0,1,5) \sqsubseteq_A$ `change`$(0,_,_)$;
- `[1, 0, 7]` `change`$(0,1,5) \not\sqsubseteq_A \mathrm{top}($`change`$(0,_,_))$.
- `[1, 0, 7]` `change`$(0,1,5) \sqsubseteq_A$ `[1, 0, 7]` `change`;
- `[1, 0, 7]` `change`$(0,1,5) \sqsubseteq_A$ `[1, 0, 7]` `change`$(0,_,_)$;

The underscore _ is a convenient syntactic sugar to instantiate a variable by itself —or by any other of the same sort, for the fact.

2.4.3 Syntax and Semantics of TLR*

The logic TLR* is parameterized by the sets of atomic state propositions, Π_S, and spatial actions, $\mathrm{SP}(\Omega, L)$. As this set $\mathrm{SP}(\Omega, L)$ contains the atomic transition propositions, we denote it also by $\Pi_T = \mathrm{SP}(\Omega, L)$, for uniformity in notation with Π_S. The logic TLR* generalizes CTL* to allow for the use of these transition propositions. As we see later, it contains as sublogics other important temporal logics. Let us define its syntax.

There are two classes of formulas:

- formulas on computations (or paths), that are called PTLR*;
- formulas on states, that are properly called TLR*.

Here we use $\varphi, \varphi_1, \varphi_2$ as elements of TLR*, and ϕ, ϕ_1, ϕ_2 as elements of PTLR*. We denote by σ an atomic proposition on states and by τ an atomic proposition on transitions.

$$
\begin{array}{lll}
\text{TLR*:} & \varphi ::= & \top \mid \bot \mid \sigma \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \ \mathbf{A}\,\phi \mid \ \mathbf{E}\,\phi; \\
\text{PTLR*:} & \phi ::= & \varphi \mid \tau \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \\
 & & \mathbf{X}\,\phi \mid \phi_1\,\mathbf{U}\,\phi_2 \mid \phi_1\,\mathbf{R}\,\phi_2 \mid \phi_1\,\mathbf{W}\,\phi_2 \mid \ \mathbf{F}\,\phi \mid \ \mathbf{G}\,\phi.
\end{array}
$$

It is well known that some of the connectives used above are redundant. To define the semantic relations we only use $\top$, $\neg$, $\vee$, $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{A}$. The semantics of a state formula φ is given by the relation $\mathcal{R}, s_0 \models \varphi$, where $\mathcal{R}$ is a rewrite theory with equation set $E \cup A$, and s_0 is a term of sort `State` in E-canonical form. Similarly, the semantics of a path formula ϕ is given by the relation $\mathcal{R}, (s,t) \models \phi$, where (s,t) is an infinite computation in $\mathcal{R}$. A bit of additional notation before the definitions: for a computation (s,t), its suffix resulting by removing the first k elements from s and from t is denoted by $(s,t)^k$.

In what follows:

- $s_0, s_1, \ldots$ are states;
- $t_0, t_1, \ldots$ are transitions (that is, their proof terms);
- s, s', with no underscript, are finite or infinite strings of states; and
- t, t', with no underscript, are finite or infinite strings of transitions.

The relations are defined thus:

- $\mathcal{R}, s_0 \models \top$;
- $\mathcal{R}, s_0 \models \sigma \Leftrightarrow E \cup A \vdash s_0 \models \sigma = \mathit{true}$;
- $\mathcal{R}, s_0 \models \neg\varphi \Leftrightarrow \mathcal{R}, s_0 \not\models \varphi$;
- $\mathcal{R}, s_0 \models \varphi_1 \vee \varphi_2 \Leftrightarrow \mathcal{R}, s_0 \models \varphi_1$ or $\mathcal{R}, s_0 \models \varphi_2$;
- $\mathcal{R}, (s,t) \models \varphi \Leftrightarrow \mathcal{R}, s_0 \models \varphi$;
- $\mathcal{R}, (s,t) \models \tau \Leftrightarrow t_0 \sqsubseteq_A \tau$;
- $\mathcal{R}, (s,t) \models \neg\phi \Leftrightarrow \mathcal{R}, (s,t) \not\models \phi$;
- $\mathcal{R}, (s,t) \models \phi_1 \vee \phi_2 \Leftrightarrow \mathcal{R}, (s,t) \models \phi_1$ or $\mathcal{R}, (s,t) \models \phi_2$;
- $\mathcal{R}, (s,t) \models \mathbf{X}\,\phi \Leftrightarrow \mathcal{R}, (s,t)^1 \models \phi$;
- $\mathcal{R}, (s,t) \models \phi_1 \,\mathbf{U}\, \phi_2 \Leftrightarrow \exists k \in \mathbb{N},\ \mathcal{R}, (s,t)^k \models \phi_2$ and $\forall 0 \leq i < k,\ \mathcal{R}, (s,t)^i \models \phi_1$;
- $\mathcal{R}, (s,t) \models \mathbf{F}\,\phi \Leftrightarrow \exists k \in \mathbb{N}$ s. t. $\mathcal{R}, (s,t)^k \models \phi$;
- $\mathcal{R}, s_0 \models \mathbf{A}\,\phi \Leftrightarrow$ for all computations (s,t) in $\mathcal{R}$ starting at s_0, we have $\mathcal{R}, (s,t) \models \phi$.

2.4.4 Sublogics of TLR*

TLR* is a very general logic that contains other well-known and important ones. For instance, when the set of atomic action-propositions is taken to be empty, we get CTL*(Σ_S). (The usual definition of CTL* allows only for constants as atomic propositions, while we allow for more general expressions, so CTL*(Σ_S) is not exactly CTL*.)

Restricting the use of connectives we get, for instance:

- CTL and its generalizations (adding transition propositions) TLR and PTLR;
- LTL and its generalization LTLR;
- SR and GR, that generalize the quantifier-free safety and guarantee formulas in LTL.

GR formulas, that is, guarantee formulas in TLR*, are the ones that interest us the most, because our model checker accepts them as input. Remember that in LTL, and so in GR, formulas are implicitly taken to be universally path-quantified, so that they do not explicitly include **E** or **A**. Also, remember that guarantee formulas express that something (taken as good) is surely going to happen at some (not determined) time in the future.

This is GR's syntax, denoting by $\epsilon, \epsilon_1, \epsilon_2$ its formulas, and being σ and τ as above:

$$\mathrm{GR} : \epsilon ::= \top \mid \bot \mid \sigma \mid \neg\sigma \mid \tau \mid \neg\tau \mid \epsilon_1 \vee \epsilon_2 \mid \epsilon_1 \wedge \epsilon_2 \mid \mathbf{X}\,\epsilon \mid \epsilon_1 \,\mathbf{U}\, \epsilon_2 \mid \mathbf{F}\,\epsilon$$

Note, in particular, that only atoms can be negated.

2.5 A Strategy Language

A strategy language for Maude specifications is described in this section, with its syntax and semantics. This is going to play a central role in our work, as TLR* guarantee formulas are translated into strategy expressions as an intermediate step to its model checking.

2.5.1 Strategies

Rewriting systems are nondeterministic. Sometimes we need to *guide* the system's evolution to remove or restrict the nondeterminism. A strategy language can be used for that purpose. There exists a rich strategy language for Maude [19]. Below we describe, following again [21], another such strategy language, that has been designed with the construction of the model checker in sight.

A strategy expression e can be seen in two equivalent ways: the first, as already stated, as a way to guide a system; the second, as a question on whether the system is able to follow a path according to e. In this second sense, a strategy expression works pretty much like a temporal logic formula. Indeed, the model checker we present below first translates TLR* guarantee formulas into strategy expressions and, then, analyses the system according to the strategy.

2.5.2 A Strategy Language

In this section we present the syntax and semantics of the strategy language used by the model checker. The syntax depends on two parameters: the atomic state propositions and the atomic transition propositions of the system. As we did above, we denote by σ a generic atomic proposition on states and by τ a generic atomic proposition on transitions. The definition contains three syntactic categories: *Test* (tests on states), *Strat* (strategy expressions), and *StratForm* (strategy formulas). Here, e, e_1, e_2 are strategies (that is, strategy expressions), and b, b_1, b_2 are tests.

- *Test*: $b ::= \top \mid \bot \mid \sigma \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2$
- *Strat*: $e ::= \text{idle} \mid \tau \mid \neg\tau \mid \text{any} \mid e_1 \wedge e_2 \mid (e_1 \mid e_2) \mid e_1 \,;\, e_2 \mid e^+ \mid e_1 \,\mathcal{U}\, e_2 \mid e \,.\, b$
- *StratForm*: $f ::= \mathbf{A}\, e \mid \mathbf{E}\, e$

The language can be extended by the standard expressions $e^* = \text{idle} \mid e^+$ and $e^0 = \text{idle}$, $e^{n+1} = e \,;\, e^n$. A few explanations follow on the operators that may not be trivial:

- The strategy idle does nothing and leaves the system in the same state it was. Or, seen in the *temporal-formula* way, it requests nothing from the system, so that every path satisfies it.
- The expression $e_1 \,;\, e_2$ means sequential composition, that is, the system is first guided by e_1 and then, when e_1 has finished its job, by e_2. Or, seen the other way, a path satisfies $e_1 \,;\, e_2$ iff an initial segment of it satisfies e_1 and the remainder satisfies e_2.
- The strategy $e_1 \wedge e_2$ has a conjunctive meaning. However, given a finite computation (s, t), we do not insist that both e_1 and e_2 hold for the whole of (s, t): it is enough for one of them, e_1 or e_2, to hold for the whole of (s, t), and for the other to hold for an initial segment of (s, t). Seen in the *guiding* way: both e_1 and e_2 guide the system together through the same path for a while until one of them gets exhausted and, then, the system follows the other one.
- The strategy $e_1 \,\mathcal{U}\, e_2$ is an *until* operator with the expected meaning: either e_2 holds for the whole computation, or e_1 holds for subcomputations beginning at the first step, at the second, and so on, until a subcomputation beginning at state n, and then e_2 holds for a subcomputation beginning at state $n + 1$. However, as in the case for $e_1 \wedge e_2$, all these subcomputations beginning at different stages need not end exactly when the entire computation does: they could end before. It is enough to require that at least one of them ends when the whole finite computation does.
- The strategy $e \,.\, b$ combines e with a test b. It holds on a finite computation iff e holds and the test b succeeds for the last state in the computation.

A couple of examples could be in order here. For the system `COUNTING` consider the strategy

({'change} ; {'resend}* ; {'remove})*

It tells the system that each `change` must be followed by some `resend`s and a `remove`. Because of the way the system is specified, the rule `remove` is only enabled after `resend` has been taken as many times as possible. So, in practice, this is telling that a change must be reflected in all counters and removed from the system before any other change can happen. This is a sensible strategy, and is better seen as a *guiding* strategy.

For the promised second example strategy, consider

any* . `noMsgs`

where we assume that `noMsgs` is an atomic proposition. It says that sometime in the future there will be no messages in the system state. This is probably better seen as a *temporal-property* strategy.

By the way, to be used in our model checker, the proposition `noMsgs` can be defined in Maude like this:

```
op noMsgs : -> StateProp .
eq (I |> N) S |= noMsgs = false .
eq S |= noMsgs = true [owise] .
```

Here `S` is a variable of sort `State`, and `I` and `N` are `Id`s. In words: a state satisfies `noMsgs` iff it cannot match a pattern having a message. See Sections 3.4 and 3.5 for more explanations on defining state or transition properties.

2.5.3 Strategy Semantics

The semantics that follows is taken quite literally from [21]. It includes definitions for the satisfaction relation $\models$ concerning the three syntactic categories. We begin with *Test*.

- $\mathcal{R}, s_0 \models \top$
- $\mathcal{R}, s_0 \not\models \bot$
- $\mathcal{R}, s_0 \models \sigma \Leftrightarrow \sigma \in \mathcal{L}_{\mathcal{R}}(s_0)$
- $\mathcal{R}, s_0 \models \neg b \Leftrightarrow \mathcal{R}, s_0 \not\models b$
- $\mathcal{R}, s_0 \models b_1 \vee b_2 \Leftrightarrow \mathcal{R}, s_0 \models b_1$ or $\mathcal{R}, s_0 \models b_2$
- $\mathcal{R}, s_0 \models b_1 \wedge b_2 \Leftrightarrow \mathcal{R}, s_0 \models b_1$ and $\mathcal{R}, s_0 \models b_2$

The set $\mathcal{L}_{\mathcal{R}}(s_0)$ contains the atomic propositions that $\mathcal{R}$ assigns as true to s_0.

As for *StratForm*, this is its semantics:

- $\mathcal{R}, s_0 \models \mathbf{A}\, e \Leftrightarrow$ all computations beginning at s_0 have a finite prefix that satisfies e;
- $\mathcal{R}, s_0 \models \mathbf{E}\, e \Leftrightarrow$ some computation beginning at s_0 has a finite prefix that satisfies e.

Finally, let us focus on category *Strat*. All computations (s, t) in the following definition are necessarily finite; this semantics is not defined on infinite computations. When we write computations as (ss_is', tt') we are assuming that it is a *well-formed decomposition*, that is, (ss_i, t) is a valid computation finishing in state s_i, and (s_is', t') is also a valid computation starting at s_i. In this case, we accept as possible that some of the strings are nil, that is, $|s| = |t| = 0$, or $|s'| = |t'| = 0$ or both. Equipped with this, we define:

- $\mathcal{R}, (s_0, \text{nil}) \models \text{idle}$
- $\mathcal{R}, (s_0s_1, t_0) \models \tau \Leftrightarrow t_0 \sqsubseteq_A \tau$
- $\mathcal{R}, (s_0s_1, t_0) \models \neg\tau \Leftrightarrow t_0 \not\sqsubseteq_A \tau$
- $\mathcal{R}, (s_0s_1, t_0) \models \text{any}$
- $\mathcal{R}, (s, t) \models e_1 \wedge e_2 \Leftrightarrow$ there exists i such that $(s, t) = (s's_is'', t't'')$ and either $\big(\mathcal{R}, (s, t) \models e_1$ and $\mathcal{R}, (s's_i, t') \models e_2\big)$ or $\big(\mathcal{R}, (s's_i, t') \models e_1$ and $\mathcal{R}, (s, t) \models e_2\big)$
- $\mathcal{R}, (s, t) \models e_1 \,|\, e_2 \Leftrightarrow \mathcal{R}, (s, t) \models e_1$ or $\mathcal{R}, (s, t) \models e_2$
- $\mathcal{R}, (s, t) \models e_1 \,;\, e_2 \Leftrightarrow$ there exists i such that $(s, t) = (s's_is'', t't'')$ and $\mathcal{R}, (s's_i, t') \models e_1$ and $\mathcal{R}, (s_is'', t'') \models e_2$
- $\mathcal{R}, (s, t) \models e^+ \Leftrightarrow$ either $\mathcal{R}, (s, t) \models e$ or $\big($there exists i such that $(s, t) = (s's_is'', t't'')$ and $\mathcal{R}, (s's_i, t') \models e$ and $\mathcal{R}, (s_is'', t'') \models e^+\big)$
- $\mathcal{R}, (s, t) \models e_1 \,\mathcal{U}\, e_2 \Leftrightarrow$ either $\mathcal{R}, (s, t) \models e_2$ or there exists i such that $(s, t) = (s's_is'', t't'')$ and the following conditions hold:
  - $|s'| = n \geq 1$
  - there exist $k_0, \ldots, k_n, k$, satisfying inequalities $1 \leq k_j \leq |s| - j, (j = 0, \ldots, n)$, and $1 \leq k \leq |s_is''|$, with at least one of those inequalities an actual equality,
  - $\mathcal{R}, (s, t)|_{0..k_0} \models e_1$ and $\ldots$ and $\mathcal{R}, (s, t)|_{n..n+k_n} \models e_1$ and $\mathcal{R}, (s_is'', t'')|_{0..k} \models e_2$
- $\mathcal{R}, (ss_n, t) \models e \,.\, b \Leftrightarrow \mathcal{R}, (ss_n, t) \models e$ and $\mathcal{R}, s_n \models b$

Most items are straightforward, but two of them deserve some explanation. In the case $e_1 \wedge e_2$, we do not ask both components to last until the end of the computation, but just one (any) of them. Sometimes it helps to consider strategies as touristic guides that have to take you from your point of departure to some other destination point. If you hire two guides, it is enough that one of them is able to take you to the destination, as long as they agree while they are both active.

The case $e_1 \mathcal{U} e_2$ has a quite involved formulation. The notation $(s,t)|_{i..j}$ means the subcomputation of (s,t) that begins with the state at index i and ends with the state at index j, and includes all transitions in between. The condition says that guide e_2 is able to take you to the destination, but not necessarily from the start. If it is not from the start, then e_1 is able to guide you from the start, at least one step; after that one step is taken, if e_2 is not there waiting for you, e_1 is again able to guide you at least one more step; and so on, until, necessarily, e_2 is found before arriving at the destination. In this case, e_1 is not authorized to get to the destination point.

2.5.4 Guarantee Formulas as Strategies

Our model checker internally works with strategies, but we want to use TLR* guarantee formulas. Obviously, a semantically appropriate translation from the former to the latter is needed. It is given by this function $\beta : \mathrm{GR}(\Pi_S, \Pi_T) \to \mathit{Strat}(\Pi_S, \Pi_T)$. (Remember that GR is the "guarantee" part of TLR*.)

- $\beta(\epsilon) = \epsilon$ for $\epsilon = \top, \bot, \tau, \neg\tau$
- $\beta(\epsilon) = \mathrm{idle} \,.\, \epsilon$ for $\epsilon = \sigma, \neg\sigma$
- $\beta(\epsilon_1 \vee \epsilon_2) = \beta(\epsilon_1) \,|\, \beta(\epsilon_2)$
- $\beta(\epsilon_1 \wedge \epsilon_2) = \beta(\epsilon_1) \wedge \beta(\epsilon_2)$
- $\beta(\mathbf{X}\, \epsilon) = \mathrm{any} \,;\, \beta(\epsilon)$
- $\beta(\epsilon_1 \,\mathbf{U}\, \epsilon_2) = \beta(\epsilon_1) \,\mathcal{U}\, \beta(\epsilon_2)$
- $\beta(\mathbf{F}\, \epsilon) = \mathrm{any}^* \,;\, \beta(\epsilon)$

In [21, Theorem 3 and Corollary 1], Meseguer states and proves the following key results, that justify the use of strategies instead of formulas for model checking:

Theorem 2.2. *Given a computable rewrite theory $\mathcal{R}$, a formula $\epsilon \in \mathrm{GR}(\Pi_S, \Pi_T)$ and an infinite computation (s,t) in $\mathcal{R}$, we have*

- $\mathcal{R}, (s,t) \models \epsilon \Leftrightarrow \exists k \in \mathbb{N}$ *such that* $\mathcal{R}, (s,t)|_{0..k} \models \beta(\epsilon)$
- $\mathcal{R}, s_0 \models \mathbf{A}\, \epsilon \Leftrightarrow \mathcal{R}, s_0 \models \mathbf{A}\, \beta(\epsilon)$
- $\mathcal{R}, s_0 \models \mathbf{E}\, \epsilon \Leftrightarrow \mathcal{R}, s_0 \models \mathbf{E}\, \beta(\epsilon)$

2.6 Model Checking for Finite and Infinite Systems

Model checking is the task of automatically proving that a system meets a specification. For this to be possible, both the system and the specification need to be given in precise, formal languages. Model checking procedures work by exhaustive inspection of the system's state-space. Techniques have been developed to help reduce the size of the state-space: for instance, state abstraction, that identifies sets of states whose differences are not meaningful to the property we try to check. Model checking opposes and complements theorem proving.

Being based on exhaustive search, model checking is not directly amenable to be applied on infinite systems. Sometimes, abstraction can be used to reduce an infinite number of states to a finite number of abstract states. For real-time systems, continuous time can be divided into time-regions, again with the idea that the property we are seeking to prove is only able to distinguish two moments in time if they belong to different regions. More in general, model checking of infinite systems is possible by requiring the system to be finitely representable in some way or another (see, for instance, [16]).

What we present in the next chapter, following our favorite reference [21], is a procedure for explicit-state model checking on infinite systems that provides a semi-decision algorithm for guarantee formulas.

Chapter 3

The Model Checker

With all the necessary material already introduced, it is time to describe the model checker. This chapter includes some additional theoretical constructions and results and, then, details on the Maude implementation. Almost all the material in this chapter is original.

This can be a good place to remember the goal to which all what follows is directed. We will be given a system's specification and a guarantee temporal formula. First, we will translate the formula to a strategy that expresses the same guarantee property. A guarantee property is a statement that something is going to happen in the future. If it is the case that the system satisfies the guarantee property, we will find it out by searching deep enough. However, if the system does not satisfy the guarantee property, in many cases, the search could go on forever without never reaching a definitive answer. That is why we only have a semi-decision procedure.

3.1 Our TLR* Flavor

For practical reasons, and partly following [3], we do not implement our model checker for TLR* (or its guarantee formulas) exactly as described in [21], but for a slight variation of it that we want to make formal now.

There are two changes from the original, *standard* description given in Sections 2.3 and 2.4. The first is that spatial actions —that is, transition patterns— are required to be ground terms. The second is that we allow atomic transition propositions in our signature, parallel to atomic state propositions and at the same level that transition patterns.

3.1.1 Transition Proof Terms

Let $\mathcal{R} = (\Sigma, E \cup A, R)$ be a rewriting system. We define next a signature $\mathrm{Trans}(\Sigma)$ (Trans for "transition"), on which one-step proof terms are built, extending Σ in this way:

- For each sort $S \in \Sigma$, we add a new sort $\mathrm{Trans}(S)$ to $\mathrm{Trans}(\Sigma)$, and state that $S < \mathrm{Trans}(S)$, that is, S is a subsort of $\mathrm{Trans}(S)$ (there are good technical reasons to this requirement, that we do not discuss here).

- We add `Trans` as a convenient synonym for the sort Trans(`State`), the sort of transitions between terms of sort `State`.

- Given a rule $l : q \to q'$ in R, let S be the sort of q, and let the variables appearing in q, taken in their textual order of appearance, have sorts $S_1, \ldots, S_n$. Then, for each such rule $l \in R$, we add to Trans(Σ) a new function symbol $l : S_1 \times \cdots \times S_n \to \mathrm{Trans}(S)$.

- For each function symbol $f : S_1 \times \cdots \times S_n \to S$ in Σ and each $i = 1, \ldots, n$, we add to Trans(Σ) an overloaded function symbol

$$f : S_1 \times \cdots \times \mathrm{Trans}(S_i) \times \cdots \times S_n \to \mathrm{Trans}(S).$$

  The reason for this involved definition is that we need to be sure that only one of the new rule-label function symbols appears in each one-step proof term.

Now, consider the rewriting system $\mathrm{Trans}(\mathcal{R}) = (\mathrm{Trans}(\Sigma), E \cup A, R)$. There are no new equations and no new rules. Thus, if $\mathcal{R}$ was computable (as defined in Section 2.3.1), so is Trans($\mathcal{R}$). In particular, every term has a unique E-canonical form modulo A. The set $(\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{Trans}}$ is the set of all A-equivalence classes of the form $[\mathrm{can}_{E/A}(t)]_A$, where t is a ground-term of sort `Trans`. That is, $(\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{Trans}}$ describes the set of all one-step transitions between `State`s in the system specified by $\mathcal{R}$ in their canonical form representation. This is in parallel to $(\mathrm{Can}_{\Sigma/E\cup A})_{\mathtt{State}} = (\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{State}}$. This set $(\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{Trans}}$ is the same that was called $\mathrm{CanPTerms}^1(\mathcal{R})$ in Section 2.3.2.

3.1.2 Transition Patterns

As told above, we are asking action patterns to be ground terms. The reason to decide so is that action patterns that are not ground tend to be seldom used in practice. Also, as we will see, the small loss that this change supposes will be overcome by the second change we propose.

This restriction to ground terms has an advantageous side effect: we do not need to restrict our patterns to use only constructor symbols as we did in Section 2.4.2. Remember that the reason to restrict to constructor symbols was so to make decidable the "instance of" relation between a proof term and a pattern. But on ground terms and on a computable rewriting system —which $\mathrm{Pattern}(\mathcal{R}) = (\mathrm{Pattern}(\Sigma), E \cup A, R)$ still is— that relation is indeed decidable.

Let us see the signature Pattern(Σ) on which these patterns are ground terms:

- $\mathrm{Trans}(\Sigma) \subset \mathrm{Pattern}(\Sigma)$.

- For each sort $S \in \Sigma$, we add a new sort Pattern(S) to Pattern(Σ). These are the patterns for transitions between terms of sort S. We state that $\mathrm{Trans}(S) < \mathrm{Pattern}(S)$. In particular, each proof term can be used as a pattern for transitions between states.

- For each sort $S \in \Sigma$, we add a new sort Maybe(S) to Pattern(Σ). We state that $S <$ Maybe(S) and declare a new constant `_` (underscore) of sort Maybe(S). Elements of sort Maybe(S) are used to instantiate function arguments of sort S to produce patterns. The underscore is used when we do not care about the value of this particular argument in our pattern, so that, for instance, whether or not a term matches the pattern `f(0,_)` does not depend on its second argument.

- Given a rule $l : q \to q'$ in R, let S be the sort of q and let the variables appearing in q, taken in their textual order of appearance, have sorts $S_1, \dots, S_n$. Then, for each such rule $l \in R$, we add to Pattern(Σ):
  - a constant l of sort Pattern(S);
  - a constant top-l of sort Pattern(`State`);
  - a function $l : \text{Maybe}(S_1) \times \cdots \times \text{Maybe}(S_n) \to \text{Pattern}(S)$;
  - a function top-$l : \text{Maybe}(S_1) \times \cdots \times \text{Maybe}(S_n) \to \text{Pattern}(\texttt{State})$.

- For each function symbol $f : S_1 \times \cdots \times S_n \to S$ in Σ and each $i = 1, \dots, n$, we add to Pattern(Σ) an overloaded function symbol

$$f : S_1 \times \cdots \times \text{Pattern}(S_i) \times \cdots \times S_n \to \text{Pattern}(S).$$

Ground terms of sort Pattern(`State`) are the ones that can be used in TLR* formulas. We must define the notion of pattern in E-canonical form modulo A in the obvious way.

Finally, we establish when a term matches a pattern. As we use the notation $\mathcal{R}, s_0 \models \sigma$ for states and their propositions, so we use now $\mathcal{R}, t_0 \models \tau$. The definition goes like this, including just the positive cases:

- $\mathcal{R}, v[l(u_1, \dots, u_n)]_p \models l$;
- $\mathcal{R}, l(u_1, \dots, u_n) \models \text{top-}l$;
- $\mathcal{R}, v[l(u_1, \dots, u_n)]_p \models l(u'_1, \dots, u'_n)$ if $\forall i$ we have either $u'_i = _$ or $\mathcal{R} \vdash u_i = u'_i$;
- $\mathcal{R}, l(u_1, \dots, u_n) \models \text{top-}l(u'_1, \dots, u'_n)$ if $\forall i$ we have either $u'_i = _$ or $\mathcal{R} \vdash u_i = u'_i$;
- $\mathcal{R}, v[l(u_1, \dots, u_n)]_p \models v'[l]_{p'}$ if $\text{Pattern}(\mathcal{R}) \vdash v[l]_p = v'[l]_{p'}$;
- $\mathcal{R}, v[l(u_1, \dots, u_n)]_p \models v'[l(u'_1, \dots, u'_n)]_{p'}$ if $\text{Pattern}(\mathcal{R}) \vdash v[l]_p = v'[l]_{p'}$ and $\forall i$ we have either $u'_i = _$ or $\mathcal{R} \vdash u_i = u'_i$.

3.1.3 Atomic Propositions on Transitions

This is the second point in which *our* TLR* differs from the *standard*: we consider a signature of atomic propositions on transitions, in addition to the atomic propositions on states. This can be seen as a step forward in the way of taking states and transitions to the same level.

In Section 2.3.1, in order to make our lives easier, we asked our rewriting systems to have a sort `State`, a sort `StateProp`, a symbol `|=`, and so on. Now, we add some requirements related to the new signatures we use. For ease of reference, we list here the previous requirements along the new ones:

- One of the sorts in the signature must be called `State`. The sort Trans(`State`) is equivalently called `Trans`.
- $\mathcal{R}$ has a sort `Bool` with constants `true` and `false`.
- If $\mathcal{R}$ has a sort named `StateProp` (it needs not), then it must also have an operator `|=` : `State` $\times$ `StateProp` $\to$ `Bool`. Furthermore, if Σ is the signature of $\mathcal{R}$, then we define the subsignature $\Sigma_S \subseteq \Sigma$ of its state proposition symbols as the set of all operators in Σ whose result sort is `StateProp`. Also, we denote by Π_S the set of terms of sort `StateProp`.
- $\mathcal{R}$ has a sort named `TransProp` (it must), and an operator `|=` : `Trans` $\times$ `TransProp` $\to$ `Bool`. Furthermore, if Σ is the signature of $\mathcal{R}$, then we define the subsignature $\Sigma_T \subseteq$ Pattern(Σ) of its transition proposition symbols as the set of all operators in Pattern(Σ) whose result sort is `TransProp`. Also, we denote by Π_T the set of terms of sort `TransProp`.
- Pattern(`Trans`) $<$ `TransProp`. That is, transition patterns are considered as propositions on transitions, as they should. Indeed, they are treated as *atomic* propositions. Note that we previously used Π_T to denote the set of transition patterns, when they were the only atomic transition propositions we had. Thus, Π_T is still the set of all atomic transitions propositions.
- $\mathcal{R}$ must be deadlock-free.

In practice, when using our tool, as better explained below, the constants or functions in Π_T are defined in the user's system module (or in an addition to it), and so are the conditions under which they hold for a proof term. This is the same we do for atomic state propositions.

3.2 Strategy Semantics

In this section we are giving a new definition for strategy semantics, equivalent to the one in Section 2.5.3. The new definition is designed to be almost literally translatable into Maude

code. This supposes mainly that it is more operational in style, being performed step by step, in parallel to the evolution of the system.

Just as a refresher, this is the syntax for the strategy language, as defined in Section 2.5.2:

- *Test*: $b ::= \top \mid \bot \mid \sigma \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2$
- *Strat*: $e ::= \mathrm{idle} \mid \tau \mid \neg\tau \mid \mathrm{any} \mid e_1 \wedge e_2 \mid (e_1 \,|\, e_2) \mid e_1 \,;\, e_2 \mid e^+ \mid e_1 \mathcal{U}\, e_2 \mid e.b$
- *StratForm*: $f ::= \mathbf{A}\, e \mid \mathbf{E}\, e$

From the semantics given in Section 2.5.3, an equivalence relation $e_1 \equiv e_2$ can be defined:

Definition 3.1 (Semantic equivalence)**.** *For strategy expressions e_1 and e_2, we define $e_1 \equiv e_2$ by $\mathcal{R}, (s,t) \models e_1 \Leftrightarrow \mathcal{R}, (s,t) \models e_2$ for all rewriting systems $\mathcal{R}$ and for all finite computations (s,t) in them.*

Lemma 3.2. *The relation $\equiv$ is an equivalence relation and a congruence with respect to the operators of the strategy language.*

The proof is easy and is sketched on Meseguer's paper [21]. For this semantic equivalence we have the following original result:

Proposition 3.3. *Any strategy expression e is semantically equivalent to one in the following disjunctive form:*

$$\mathrm{idle} \,.\, C_1 \;\mid\; \ldots \;\mid\; \mathrm{idle} \,.\, C_m \;\mid\; \mathrm{idle} \,.\, B_1 \,;\, T_1 \,;\, e_1 \;\mid\; \ldots \;\mid\; \mathrm{idle} \,.\, B_n \,;\, T_n \,;\, e_n$$

where

- *for $i = 1, \ldots, \max(m,n)$, each B_i and each C_i are either $\bot$ or conjunctions of atomic state propositions and their negations: $\sigma_{i,1} \wedge \cdots \wedge \sigma_{i,n_i} \wedge \neg\sigma_{i,n_i+1} \wedge \cdots \wedge \neg\sigma_{i,n_i+n'_i}$;*
- *for $i = 1, \ldots, n$, each T_i is a conjunction of atomic transition propositions and their negations: $T_i = \tau_{i,1} \wedge \ldots \wedge \tau_{i,m_i} \wedge \neg\tau_{i,m_i+1} \wedge \ldots \wedge \neg\tau_{i,m_i+m'_i}$;*
- *for $i = 1, \ldots, n$, each e_i is a general strategy expression.*

In particular, we admit empty conjunctions and disjunctions, with the conventions:

- *if $n_i = n'_i = 0$, then $B_i = \top$, resp. $C_i = \top$;*
- *if $m_i = m'_i = 0$, then $T_i = \mathrm{any}$;*
- *if $n = 0$, then the whole expression gets reduced to $\mathrm{idle} \,.\, C_1 \mid \ldots \mid \mathrm{idle} \,.\, C_m$;*
- *if $m = 0$, then the expression gets reduced to $\mathrm{idle} \,.\, B_1 \,;\, T_1 \,;\, e_1 \mid \ldots \mid \mathrm{idle} \,.\, B_n \,;\, T_n \,;\, e_n$.*

Proof. By structural induction. Let us sketch all cases.

Base cases are easy:

- $\text{idle} \equiv \text{idle} . \top$;
- $\tau \equiv \text{idle} . \top \,;\, \tau \,;\, \text{idle}$;
- and so on.

Regarding the operator '$\wedge$', we need to use the equivalence

$$e_1 \wedge (e_2 \,|\, e_3) \equiv (e_1 \wedge e_2) \,|\, (e_1 \wedge e_3),$$

whose proof is easy and is omitted. By applying repeatedly that equivalence, and also the commutativity and associativity of $\wedge$ and $|$, we can transform $(e_1 \,|\, e_2 \,|\, \dots) \wedge (e'_1 \,|\, e'_2 \,|\, \dots)$ into $(e_1 \wedge e'_1) \,|\, (e_1 \wedge e'_2) \,|\, \dots$. Thus, we only need to show how $\wedge$ works at the lowest level:

- $\text{idle} . C_1 \wedge \text{idle} . C_2 \equiv \text{idle} . (C_1 \wedge C_2)$;
- $\text{idle} . C_1 \wedge (\text{idle} . B_1 \,;\, T_1 \,;\, e_1) \equiv \text{idle} . (C_1 \wedge B_1) \,;\, T_1 \,;\, e_1$;
- $(\text{idle} . B_1 \,;\, T_1 \,;\, e_1) \wedge (\text{idle} . B_2 \,;\, T_2 \,;\, e_2) \equiv \text{idle} . (B_1 \wedge B_2) \,;\, (T_1 \wedge T_2) \,;\, (e_1 \wedge e_2)$.

Note that all the terms on the right-hand sides of the equivalences have the necessary form. All these equivalences are easily proven correct.

The proof for the operator '$|$' is very easy. So is the '.' (dot operator). For the operator '$;$' we need to use the equivalence $e_1 \,; (e_2 \,|\, e_3) \equiv (e_1 \,;\, e_2) \,|\, (e_1 \,;\, e_3)$.

The case $e_1 \mathcal{U} e_2$ is easily unfolded to $e_2 \,|\, (e_1 \wedge (\text{any} \,; (e_1 \mathcal{U} e_2)))$. The induction hypothesis applied to e_1 and e_2 ensures that they can be written in the disjunctive form. The other operand, $\text{any} \,; (e_1 \mathcal{U} e_2)$, is equivalent to $\text{idle} . \top \,; \text{any} \,; (e_1 \mathcal{U} e_2)$, that has the form of one disjunct. As the operators '$\wedge$' and '$|$' have been dealt with above, the case $e_1 \mathcal{U} e_2 \equiv e_2 \,|\, (e_1 \wedge (\text{any} \,; (e_1 \mathcal{U} e_2)))$ is complete.

The operator '$^+$' deserves some more attention. Let us take a simple example (the general one follows the same lines of reasoning): $(\text{idle} . C \,|\, \text{idle} . B \,;\, T \,;\, e)^+$. One possibility is that, each time we make a choice, it is $\text{idle} . C$ what we choose. Then we get $(\text{idle} . C)^+ = \text{idle} . C$. Otherwise, at some point we choose $\text{idle} . B \,;\, T \,;\, e$ (let us suppose it is not at the very beginning) and get as result $\text{idle} . (C \wedge B) \,;\, T \,;\, \big(e \,; (\text{idle} . C \,|\, \text{idle} . B \,;\, T \,;\, e)^+\big)$. Both results have the correct form. □

Note that when $n = 0$, only zero-step computations can satisfy the strategy, that is, computations of the form (s_0, nil). On the other hand, when $m = 0$, only computations that do perform at least one step can satisfy the strategy.

Definition 3.4. *A* zero-step strategy *is one that can only be satisfied by zero-step computations like* (s_0, nil). *A* nonzero-step strategy *is one that can only be satisfied by nonzero-step computations like* $(s_0 s_1 s, t_0 t)$.

In general, a strategy need not be zero-step nor nonzero-step.

Proposition 3.3 provides the idea to our semantics definition: a computation satisfies a strategy if either the starting state satisfies some of the zero-step conditions, or the first step satisfies some of the nonzero-step conditions and the rest of the computation satisfies the rest of the strategy. Written in an informal way:

$$\begin{aligned}
&\mathcal{R}, (s_0 s_1 s, t_0 t) \models \text{idle} \,.\, C_1 \;\mid\; \ldots \;\mid\; \text{idle} \,.\, B_1 \,;\, T_1 \,;\, e_1 \;\mid\; \cdots \Leftrightarrow \\
&\qquad \mathcal{R}, s_0 \models C_1 \text{ or } \;\ldots \\
&\qquad \text{or} \\
&\qquad \Big(\mathcal{R}, s_0 \models B_1 \text{ and } \mathcal{R}, t_0 \models T_1 \text{ and } \mathcal{R}, (s_1 s, t) \models e_1\Big) \text{ or } \;\ldots
\end{aligned}$$

This allows us to explore the evolution of the system step by step and make the strategy evolve in parallel deciding, at each step, whether the strategy has already been fully satisfied or fully falsified or we have to go on.

We could surely find and code a function that transforms any strategy expression to its disjunctive form and define a semantics from it. However, this does not seem to provide the best result from a computational complexity point of view. What we want to do, instead, is to simplify, or *advance*, the strategy expression, at each step, *applying* to it the knowledge we have about the current state and the current transition, scanning the strategy expression as it is. We need some preparation.

Lemma 3.5. *The following equations are correct with respect to the semantic equivalence defined above:*

- $e \wedge e = e$;
- $\text{idle} \wedge e = e$;
- $\text{idle} \,.\, b_1 \wedge \text{idle} \,.\, b_2 = \text{idle} \,.(b_1 \wedge b_2)$;
- $e \,|\, e = e$;
- $\text{idle} \,|\, \text{idle} \,.\, b = \text{idle}$;
- $\text{idle} \,.\, b_1 \,|\, \text{idle} \,.\, b_2 = \text{idle} \,.(b_1 \vee b_2)$;
- $\text{idle} \,;\, e = e$;
- $e \,;\, \text{idle} = e$;
- $\text{idle} \,.\, \top = \text{idle}$;
- $\text{idle} \,.\, b_1 \,;\, \text{idle} \,.\, b_2 = \text{idle} \,.(b_1 \wedge b_2)$;
- $(\text{idle} \,.\, b_1) \,.\, b_2 = \text{idle} \,.(b_1 \wedge b_2)$;
- $\text{idle} \,.\, \bot \,\mathcal{U}\, e = e$;

- $\text{idle}^+ = \text{idle}$*;*

- $(\text{idle} \,.\, b)^+ = \text{idle} \,.\, b$*;*

- $e_1 \wedge (e_2 \,|\, e_3) = (e_1 \wedge e_2) \,|\, (e_1 \wedge e_3)$*;*

- $e_1 \,;\, (e_2 \,|\, e_3) = (e_1 \,;\, e_2) \,|\, (e_1 \,;\, e_3)$*;*

- $(e_1 \,|\, e_2) \,;\, e_3 = (e_1 \,;\, e_3) \,|\, (e_2 \,;\, e_3)$*;*

- $(e_1 \,|\, e_2) \,.\, b = (e_1 \,.\, b) \,|\, (e_2 \,.\, b)$*.*

Commutativity and associativity for '∧' and '|' also hold, as does associativity for '; '.

We do not include commutativity and associativity in the list as they are nonterminating. In our Maude implementation they are included as operator's attributes, so that Maude uses specific matching algorithms for them.

These equations have easy proofs. They have been chosen to make correct our definition of the semantics below. Some other such equations exist, and they are indeed used in our implementation to improve performance. But the correctness of the definition depends only on the ones listed.

Lemma 3.6. *The set of equations above is terminating and confluent modulo commutativity and associativity of '∧' and '|' and associativity of '; '.*

Proof. Termination follows if we can define an ordering on the set of strategy expressions that decreases with the left-to-right use of any of the equations and that has an absolute lower bound. For instance:

- the sum of the depths at which operators '|' appear decreases in the some equations and is constant in others;

- in the ones that the former is constant, the number of strategy operators in the expression decreases.

In both items we work with natural numbers, that, of course, cannot decrease beyond zero.

About confluence, there is a tool for Maude, called the CRC, or Church-Rosser Checker [13], that does exactly what we need. We have coded all the equations above (that was really a literal translation) in a Full Maude module called `ISMC-EQS`. Then, after loading the CRC, we typed the command:

```
(check Church-Rosser ISMC-EQS .)
```

whose answer was

```
Church-Rosser checking of ISMC-EQS
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

This proves local confluence that, in the presence of termination, implies global confluence (Newman's lemma). □

Lemma 3.7. *The set of equations above, plus a terminating and confluent set of equations for Boolean algebras (to be used on Tests), are enough to prove or disprove the following equivalences:*

- $e \equiv \text{idle}$*;*
- $e \equiv \text{idle} \,.\, B$*;*
- $e \equiv \text{idle} \,|\, \hat{e}$*;*
- $e \equiv \text{idle} \,.\, B \,|\, \hat{e}$*;*
- $e \equiv \hat{e}$*.*

That is, if any of these semantic equivalences do hold, the equations reduce e *to an irreducible expression with the shape on the right of one of the items, where* B *is a state proposition, and* $\hat{e}$ *a nonzero-step strategy expression. Beware that we do not ask* B *and* $\hat{e}$ *to have any particular form. The last case must be seen as a way to tell that, if* e *is not equivalent to any of the four first forms, then it is a nonzero-step strategy.*

Proof. The proof is again by structural induction. It is not fun to write nor read, but let us review some cases. Note that one of the directions is always true. For instance, if e is reducible to idle, as the equations are correct, then $e \equiv \text{idle}$.

We skip the easy base cases idle, τ, $\neg\tau$, and any.

First, we prove that if $e_1 \wedge e_2 \equiv \text{idle} \,.\, B$ then $e_1 \wedge e_2$ has an irreducible form (according to the equations above) like $\text{idle} \,.\, B'$. So, suppose $e_1 \wedge e_2 \equiv \text{idle} \,.\, B$. That means that $\mathcal{R}, (s,t) \models e_1 \wedge e_2$ iff $\mathcal{R}, (s,t) \models \text{idle} \,.\, B$. But $\mathcal{R}, (s,t) \models \text{idle} \,.\, B$ is only possible when $s = s_0$, $t = \text{nil}$, and $\mathcal{R}, s_0 \models B$. So we must have $\mathcal{R}, (s_0, \text{nil}) \models e_1 \wedge e_2$. By definition this is the same as $\mathcal{R}, (s_0, \text{nil}) \models e_1$ and $\mathcal{R}, (s_0, \text{nil}) \models e_2$. These conditions only hold when $e_1 \equiv \text{idle} \,.\, B_1$ and $e_1 \equiv \text{idle} \,.\, B_2$, for some tests B_1 and B_2. By structural induction hypothesis, e_1 has an irreducible form like $\text{idle} \,.\, B_1'$ and e_2 has $\text{idle} \,.\, B_2'$. Thus, $e_1 \wedge e_2$ is reducible to $\text{idle} \,.\, B_1' \wedge \text{idle} \,.\, B_2'$. Finally, by using equation $\text{idle} \,.\, b_1 \wedge \text{idle} \,.\, b_2 = \text{idle} \,.(b_1 \wedge b_2)$ from the set of equations above, we get that $e_1 \wedge e_2$ is reducible to $\text{idle} \,.(B_1' \wedge B_2')$, and this last expression can only be further reduced within the test part, keeping the same form $\text{idle} \,.\, B$.

For another case, let us suppose $e_1 \,;\, e_2 \equiv \text{idle} \,.\, B \,|\, \hat{e}$. So, $e_1 \,;\, e_2$ must be ready to accept both some zero-step computation and some nonzero-step one. This is only possible if none e_1 or e_2 are nonzero-step, and one of them is not zero-step. The most complex of these cases is $e_1 \equiv \text{idle} \,.\, B_1 \,|\, \hat{e}_1$ and $e_2 \equiv \text{idle} \,.\, B_2 \,|\, \hat{e}_2$. Using the distributive laws for '$;$' and '$|$' we get $e_1 \,;\, e_2 \equiv \text{idle} \,.(B_1 \wedge B_2) \,|(\text{idle} \,.\, B_1 \,;\, \hat{e}_2) \,|(\hat{e}_1 \,;\, \text{idle} \,.\, B_2) \,|(\hat{e}_1 \,;\, \hat{e}_2)$, that is already in the prescribed form.

The case $e_1 \,\mathcal{U}\, e_2$, being equivalent to $e_2 \,|(e_1 \wedge (\text{any} \,;(e_1 \,\mathcal{U}\, e_2)))$, is always ready to accept a step, because of the 'any' appearing in it, unless $e_1 \equiv \text{idle} \,.\, \bot$. So the only way that $e_1 \,\mathcal{U}\, e_2$ can be equivalent to $\text{idle} \,.\, B$ is that, in addition to $e_1 \equiv \text{idle} \,.\, \bot$, we have $e_2 \equiv \text{idle} \,.\, B$. □

In further preparation for the definition of our semantics, we next define a function adv (for *advance*) that simplifies a strategy by applying to it the knowledge from a state and a transition. That is, adv answers the question: For strategy e to hold from a given state s_0 on, what strategy, derived from e, must hold from s_1 on, after taking the step (s_0, t_0)? The process of advancing has two phases:

Definition 3.8. *The function* adv : *Strat*$\times$`State`$\times$`Trans` $\to$ *Strat is defined as*

$$\mathrm{adv}(e, s_0, t_0) = \mathrm{advTrans}(\mathrm{advState}(e, s_0), t_0).$$

That is, we first apply the knowledge from the state, and then from the transition.

The function advState, thus, simplifies a strategy according to whether the initial state satisfies or not the tests that appear in the *front line* of the strategy. For instance, in a simple case, $\mathrm{advState}(\mathrm{idle}\,.\,b, s_0)$ reduces to either idle or $\mathrm{idle}\,.\,\bot$ depending on whether $\mathcal{R}, s_0 \models b$ or not; but $\mathrm{any}\,.\,b$ cannot be simplified just looking at s_0, because here b has to be satisfied, not in s_0, but in the next state.

Definition 3.9. *The function* advState : *Strat*$\times$`State` $\to$ *Strat is defined by:*

- $\mathrm{advState}(\mathrm{idle}, s_0) = \mathrm{idle}$;
- $\mathrm{advState}(\tau, s_0) = \tau$;
- $\mathrm{advState}(\neg\tau, s_0) = \neg\tau$;
- $\mathrm{advState}(\mathrm{any}, s_0) = \mathrm{any}$;
- $\mathrm{advState}(e_1 \wedge e_2, s_0) = \mathrm{advState}(e_1, s_0) \wedge \mathrm{advState}(e_2, s_0)$;
- $\mathrm{advState}(e_1 \,|\, e_2, s_0) = \mathrm{advState}(e_1, s_0) \,|\, \mathrm{advState}(e_2, s_0)$;
- *for* $\mathrm{advState}(e_1\,;\,e_2, s_0)$, *compute first* $\mathrm{advState}(e_1, s_0)$ *and reduce it according to the equations in Lemma 3.5, then, to each resulting form (to the left of* $\leadsto$ *below), we define* $\mathrm{advState}(e_1\,;\,e_2, s_0)$ *as follows (to the right of* $\leadsto$*):*
  - $\mathrm{idle} \leadsto \mathrm{advState}(e_2, s_0)$;
  - $\mathrm{idle}\,.\,b \leadsto$ *if* $\mathcal{R}, s_0 \models b$ *then* $\mathrm{advState}(e_2, s_0)$ *else* $\mathrm{idle}\,.\,\bot$;
  - $\mathrm{idle} \,|\, \hat{e} \leadsto \mathrm{advState}(e_2, s_0) \,|(\hat{e}\,;\,e_2)$;
  - $\mathrm{idle}\,.\,b \,|\, \hat{e} \leadsto$ *if* $\mathcal{R}, s_0 \models b$ *then* $\mathrm{advState}(e_2, s_0) \,|(\hat{e}\,;\,e_2)$ *else* $\hat{e}\,;\,e_2$;
  - $\hat{e} \leadsto \hat{e}\,;\,e_2$.
- *for* $\mathrm{advState}(e^+, s_0)$, *compute first* $\mathrm{advState}(e, s_0)$ *and reduce it according to the equations in Lemma 3.5, then, to each resulting form (to the left of* $\leadsto$ *below), we define* $\mathrm{advState}(e^+, s_0)$ *as follows (to the right of* $\leadsto$*):*

- $\text{idle} \rightsquigarrow \text{idle}$;
- $\text{idle} \,.\, b \rightsquigarrow$ *if* $\mathcal{R}, s_0 \models b$ *then* idle *else* $\text{idle} \,.\, \bot$;
- $\text{idle} \,|\, \hat{e} \rightsquigarrow (\text{idle} \,|\, \hat{e}) \,;\, (\text{idle} \,|\, e^+)$;
- $\text{idle} \,.\, b \,|\, \hat{e} \rightsquigarrow$ *if* $\mathcal{R}, s_0 \models b$ *then* $(\text{idle} \,|\, \hat{e}) \,;\, (\text{idle} \,|\, e^+)$ *else* $\hat{e} \,;\, (\text{idle} \,|\, e^+)$;
- $\hat{e} \rightsquigarrow \hat{e} \,;\, (\text{idle} \,|\, e^+)$.

- $\text{advState}(e_1 \,\mathcal{U}\, e_2, s_0) = \text{advState}(e_2 \,|(e_1 \wedge(\text{any} \,;\, (e_1 \,\mathcal{U}\, e_2))), s_0)$;

- *for* $\text{advState}(e \,.\, b, s_0)$, *compute first* $\text{advState}(e, s_0)$ *and reduce it according to the equations in Lemma 3.5, then, to each resulting form (to the left of* $\rightsquigarrow$ *below), we define* $\text{advState}(e \,.\, b, s_0)$ *as follows (to the right of* $\rightsquigarrow$*):*
  - $\text{idle} \rightsquigarrow$ *if* $\mathcal{R}, s_0 \models b$ *then* idle *else* $\text{idle} \,.\, \bot$;
  - $\text{idle} \,.\, b' \rightsquigarrow$ *if* $\mathcal{R}, s_0 \models b \wedge b'$ *then* idle *else* $\text{idle} \,.\, \bot$;
  - $\text{idle} \,|\, \hat{e} \rightsquigarrow$ *if* $\mathcal{R}, s_0 \models b$ *then* $\text{idle} \,|(\hat{e} \,.\, b)$ *else* $\hat{e} \,.\, b$;
  - $\text{idle} \,.\, b' \,|\, \hat{e} \rightsquigarrow$ *if* $\mathcal{R}, s_0 \models b \wedge b'$ *then* $\text{idle} \,|(\hat{e} \,.\, b)$ *else* $\hat{e} \,.\, b$;
  - $\hat{e} \rightsquigarrow \hat{e} \,.\, b$.

Now we can strengthen a bit the result in Lemma 3.7 with the following easy lemma:

Lemma 3.10. *For any strategy expression e and any state s_0 we have that* $\text{advState}(e, s_0)$ *can be reduced by the set of equations in Lemma 3.5 to one of these forms:*

- idle;
- $\text{idle} \,.\, \bot$;
- $\text{idle} \,|\, \hat{e}$;
- $\hat{e}$.

Moreover, $\hat{e}$ is nonzero-step in the two last cases and:

- *if $\hat{e} = \hat{e}_1 \wedge \hat{e}_2$, then either $\hat{e}_1$ or $\hat{e}_2$ are nonzero-step;*
- *if $\hat{e} = \hat{e}_1 \,|\, \hat{e}_2$, then both $\hat{e}_1$ and $\hat{e}_2$ are nonzero-step;*
- *if $\hat{e} = \hat{e}_1 \,;\, \hat{e}_2$, then $\hat{e}_1$ is nonzero-step;*
- *if $\hat{e} = (\hat{e}_1)^+$, then $\hat{e}_1$ is nonzero-step;*
- *if $\hat{e} = \hat{e}_1 \,\mathcal{U}\, \hat{e}_2$, then $\hat{e}_2$ is nonzero-step;*
- *if $\hat{e} = \hat{e}_1 \,.\, b$, then $\hat{e}_1$ is nonzero-step.*

Proof. The first part is immediate. Most items in the "moreover" part always hold, even if $\hat{e}$ is not the result of any particular function, but just any strategy known to be nonzero-step.

For the case $\hat{e} = \hat{e}_1 \,;\, \hat{e}_2$, if $\hat{e}_1$ were idle or $\mathrm{idle} \,.\, b$, it would have been simplified by the equations in Lemma 3.5 and the function advState. If $\hat{e}_1$ were $\mathrm{idle} \,|\, \hat{e}_3$, then $\hat{e} = \mathrm{idle} \,|(\mathrm{idle} \,|\, \hat{e}_3)$ would have been reduced by associativity of '|' and equation $e \,|\, e = e$.

Other cases are similar. □

The importance of the previous lemma is that it makes possible the following definition of advancing for a transition:

Definition 3.11. *The function* advTrans : $\mathit{Strat} \times \mathtt{Trans} \to \mathit{Strat}$ *is defined by:*

- $\mathrm{advTrans}(\mathrm{idle}, t_0) = \mathrm{idle}$;
- $\mathrm{advTrans}(\tau, t_0) = \mathit{if}\ \mathcal{R}, t_0 \models \tau\ \mathit{then}\ \mathrm{idle}\ \mathit{else}\ \mathrm{idle} \,.\, \bot$;
- $\mathrm{advTrans}(\neg\tau, t_0) = \mathit{if}\ \mathcal{R}, t_0 \models \tau\ \mathit{then}\ \mathrm{idle} \,.\, \bot\ \mathit{else}\ \mathrm{idle}$;
- $\mathrm{advTrans}(\mathrm{any}, t_0) = \mathrm{idle}$;
- $\mathrm{advTrans}(e_1 \wedge e_2, t_0) = \mathrm{advTrans}(e_1, t_0) \wedge \mathrm{advTrans}(e_2, t_0)$;
- $\mathrm{advTrans}(e_1 \,|\, e_2, t_0) = \mathrm{advTrans}(e_1, t_0) \,|\, \mathrm{advTrans}(e_2, t_0)$;
- $\mathrm{advTrans}(e_1 \,;\, e_2, t_0) = \mathrm{advTrans}(e_1, t_0) \,;\, e_2$;
- $\mathrm{advTrans}(e^+, t_0) = \mathrm{advTrans}(e, t_0) \,; (\mathrm{idle} \,|\, e^+)$;
- $\mathrm{advTrans}(e_1 \,\mathcal{U}\, e_2, t_0) = \mathrm{advTrans}(e_2 \,|(e_1 \wedge (\mathrm{any} \,; (e_1 \,\mathcal{U}\, e_2))), t_0)$;
- $\mathrm{advTrans}(e \,.\, b, t_0) = \mathrm{advTrans}(e, t_0) \,.\, b$.

Except for idle, the function advTrans really advances a step, in a sense that we hope is clear. Using the function adv, we are finally in disposition to define semantics for *Strat* and the other syntactic categories:

Definition 3.12. *Let* $\mathcal{R}$ *be a rewriting system,* (s, t) *a finite or infinite computation in* $\mathcal{R}$*, and* e *a strategy expression.*

The semantics for Test does not change from the stated in Section 2.5.3: it just follows the usual semantics for Boolean operators.

For Strat, the relation $\mathcal{R}, (s, t) \models e$ *holds when some prefix of* (s, t) *satisfies* e*. Formally:*

- $$\mathcal{R}, (s_0 s_1 s, t_0 t) \models e = \begin{cases} \mathit{if}\ e \equiv \mathrm{idle} & \to \mathit{true} \\ \mathit{if}\ e \equiv \mathrm{idle} \,.\, b & \to \mathcal{R}, s_0 \models b \\ \mathit{if}\ e \equiv \mathrm{idle} \,|\, \hat{e} & \to \mathit{true} \\ \mathit{if}\ e \equiv \mathrm{idle} \,.\, b \,|\, \hat{e} & \to \mathcal{R}, s_0 \models b\ \mathit{or}\ \mathcal{R}, (s_1 s, t) \models \mathrm{adv}(\hat{e}, s_0, t_0) \\ \mathit{otherwise} & \to \mathcal{R}, (s_1 s, t) \models \mathrm{adv}(e, s_0, t_0) \end{cases}$$

- $\mathcal{R}, (s_0, \text{nil}) \models e = \begin{cases} \textit{if } e \equiv \text{idle} & \rightarrow \textit{true} \\ \textit{if } e \equiv \text{idle} \,.\, b & \rightarrow \mathcal{R}, s_0 \models b \\ \textit{if } e \equiv \text{idle} \,|\, \hat{e} & \rightarrow \textit{true} \\ \textit{if } e \equiv \text{idle} \,.\, b \,|\, \hat{e} & \rightarrow \mathcal{R}, s_0 \models b \\ \textit{otherwise} & \rightarrow \textit{false} \end{cases}$

Note that adv *always receives as parameter a nonzero-step strategy.*

Finally, the semantics for StratForm: a universally (resp. existentially) quantified strategy formula holds for a state iff all (resp. some) computations starting in that state satisfy the strategy.

We need one more lemma.

Lemma 3.13. *For convenience, we extend the definition of the* adv *function to finite computations (instead of just single steps):*

$$\text{adv}(e, s_0 s_1 s, t_0 t) = \text{adv}(\text{adv}(e, s_0, t_0), s_1 s, t).$$

Then, $\mathcal{R}, (s, t) \models e$ *iff for some* $i \in \mathbb{N}$ *we have that* $(s, t) = (s' s_i s'', t' t'')$ *and* $\text{adv}(e, s' s_i, t')$ *has one of the "success" forms:* idle, $\text{idle} \,.\, B$, $\text{idle} \,|\, \hat{e}$, $\text{idle} \,.\, B \,|\, \hat{e}$, *with* $\mathcal{R}, s_i \models B$.

The proof is easy and is skipped. Now, the key result that justifies the use of this step-wise strategy in our implementation of the model checker:

Theorem 3.14. *The two semantics for StratForm, defined in Section 2.5.3 and in Definition 3.12, coincide.*

Note that the two semantics for *Strat* cannot be directly compared, as one was defined only on finite computations and the other on possibly infinite ones.

Proof. We denote, just in this proof, the original semantics as $\models$ and the new one as $\models'$. This is once again a proof by structural induction. Let us review some cases.

For the idle, universal case we have to prove that $\mathcal{R}, s_0 \models \forall\, \text{idle} \Leftrightarrow \mathcal{R}, s_0 \models' \forall\, \text{idle}$. The $\models$ part says that each computation starting at s_0 has a finite computation that satisfies idle, that is quite trivially true. The $\models'$ part is also trivially true, just looking at the definition. Indeed, all the base cases are very easy, as usual.

Consider the '$\wedge$', universal case. We have to prove $\mathcal{R}, s_0 \models \forall e_1 \wedge e_2 \Leftrightarrow \mathcal{R}, s_0 \models' \forall e_1 \wedge e_2$. Now, the $\models$ part means that each computation $(s_0 s, t)$ can be decomposed as $(s_0 s, t) = (s_0 s' s_i s'', t' t_i t'')$ in such a way that either

$$\mathcal{R}, (s_0 s, t) \models e_1 \text{ and } \mathcal{R}, (s_0 s' s_i, t' t_i) \models e_2,$$

or the same swapping e_1 and e_2. As both cases are equal, let us consider the one we have displayed. By induction hypothesis, $\text{adv}(e_2, s_0 s' s_i, t' t_i)$ is equivalent to a "success" form, say,

for instance, $\mathrm{adv}(e_2, s_0s's_i, t't_i) \equiv \mathrm{idle} \,|\, \hat{e}_2$. Also by induction hypothesis, e_1 has a "success" form, for instance, $\mathrm{adv}(e_1, s_0s, t) \equiv \mathrm{idle} \,|\, \hat{e}_1$. Then,

$$\begin{aligned} \mathrm{adv}(e_1 \wedge e_2, s_0s's_i, t't_i) &= \mathrm{adv}(e_1, s_0s's_i, t't_i) \wedge \mathrm{adv}(e_2, s_0s's_i, t't_i) \equiv \\ &\equiv \mathrm{adv}(e_1, s_0s's_i, t't_i) \wedge (\mathrm{idle} \,|\, \hat{e}_2). \end{aligned}$$

The distributivity of adv over $\wedge$ and other operators is easily proved. We advance further:

$$\begin{aligned} \mathrm{adv}(e_1 \wedge e_2, s_0s, t) &= \mathrm{adv}(e_1 \wedge e_2, s_0s's_is'', t't_it'') = \\ &= \mathrm{adv}(e_1, s_0s, t) \wedge \mathrm{adv}(\mathrm{idle} \,|\, \hat{e}_2, s_is'', t'') = \\ &= (\mathrm{idle} \,|\, \hat{e}_1) \wedge (\mathrm{idle} \,|\, \mathrm{adv}(\hat{e}_2, s_is'', t'')) \\ &= (\mathrm{idle} \,|\, \hat{e}_1) \wedge (\mathrm{idle} \,|\, \hat{e}_2'). \end{aligned}$$

That can be readily transformed into a successful form that shows the desired result.

The rest of the cases are equally boring. □

3.3 The Main Loop

With all the developments above, we are ready to code the main loop for our model-checking algorithm, that is, the one that actually performs the search in the system's state space, building it step by step, making the strategy evolve at the same time, and checking at each step whether a final result has been reached or not.

Maude has reflective capabilities through its *metalevel* [10]. That means, for instance, that using the Maude language we can ask Maude itself about the possible rewritings from a given state, not telling Maude to actually perform those rewritings, but just to inform us about them, so that we can manipulate them in our code. We use for this the `metaXapply` function. The data that function produces includes the description of a transition from the given state and the description of the state to which that transition takes us. So, it is natural that our implementation considers this couple (transition, destination state) at the same time. We are remarking this, because the semantics we defined for the strategies followed the other approach: in the computation (s_0s_1s, t_0t) it is the couple (s_0, t_0) that is considered to have some kind of unity, that is, the *starting* state and a transition from it. Fortunately, this does not cause us much trouble.

Functions adv, advState, and advTrans are coded in Maude as direct translations from their definitions. Also, the set of equations in Definition 3.12 are translated to Maude syntax. The algorithm is based on two mutually recursive functions that we call `checkFromStep` and `checkChildren`.

The function `checkFromStep` receives a transition (its proof term) and a state and checks whether all computations starting in that transition and state, both included, satisfy the strategy. For that, it checks whether the given step is enough to satisfy or falsify the strategy and, otherwise, calls `checkChildren` to perform the remaining checking.

The function `checkChildren` receives a state, generates all steps from it using `metaXapply` and calls `checkFromStep` for each of them. Our algorithm, of course, has to take care of a few points not mentioned yet:

- whether the quantifier is `forall` or `exists` to stop or not the search when it finds a satisfying or falsifying node;
- whether the specified depth has been reached;
- storing the open branches of the computation tree;
- storing the path to the current state if loop detection is enabled;
- storing the path to the current state to report witnesses or counterexamples;
- producing or not context information according to whether context use is enabled.

The following shows a scheme of both functions, not considering the storing of open branches, nor loop detection, nor the matter with contexts. In them, D is a natural number that represents the depth that must be explored; (T, S) is a (transition, destination state) couple; Q is the path quantifier, `forall` or `exists`; and E is the strategy expression.

```
checkFromStep(D, (T, S), Q, E) =
   let E' := adv(E, T, S) in
   if E' == success then answer "Yes" (with witness if Q == exists)
   else if E' == failure then answer "No" (with witness if Q == forall)
   else if D == 0 then answer "I don't know" (and store open branches)
   else checkChildren(D-1, S, Q, E').

checkChildren(D, S, Q, E) =
   let Rdo := nothing-as-yet in
   for each child (T' ,S') of S do
      let R := checkFromStep(D, (T', S'), Q, E) in
      if R is a definitive "Yes" or "No" then answer R and return
      else let Rdo := combine(Rdo, R)
   endfor
   answer Rdo.
```

A few explanations: We consider a `success` a strategy of the form idle or idle $|\, e$; a `failure` is idle $.\, \bot$. A `Yes` answer is definitive if the quantifier is an `exists`, but not otherwise; similarly, a `No` answer is definitive if the quantifier is a `forall`, but not otherwise. The `combine` function does different things depending on the nature of the arguments. For instance, if both arguments are answers of the "I don't know" kind, each carrying its tree of open branches, `combine` joins the two trees.

Finally, the main function, the one called when the user types a model-checking command, is named `check`. It is very much like `checkFromStep`, except that at the beginning we don't have a step, but only an initial state with no transition leading to it. So the function `check` uses `let E' := advState(E, S)`, the rest being as in `checkFromStep`.

3.4 Some Notes on the Implementation

Our model checker has been implemented extending Full Maude. Full Maude is, basically, a reimplementation of Maude using Maude itself, instead of the original C++. Extending Full Maude is a usual way to develop tools that, like our model checker, are designed to act on Maude modules, because Full Maude provides facilities to internally modify user modules, add new types of modules, add commands,. . .

3.4.1 EXTENDED

Our extension has two main components: some new commands related to model checking (explained in the next section), and an operator on modules, call `EXTENDED`, that we describe now.

Given a system module —let us call it `UserMod`— our tool is able to generate the module `EXTENDED[UserMod]` and put it at the user's disposal. The `EXTENDED` operator implements what in the theoretical development of Section 3.1.2 was called Pattern. This module `EXTENDED[UserMod]` adds to `UserMod` the following:

On satisfaction

- `EXTENDED` assumes and needs that `UserMod` has a sort named `State`.
- It declares new sorts `Trans`, `StateProp`, and `TransProp`.
- It declares satisfaction operators

```
op |= : State StateProp -> Bool .
op |= : Trans TransProp -> Bool .
```

On contexts

- For each sort `S` in `UserMod`, it adds a new sort named `Context$S` and states that `S < Context$S`. It adds a new sort `UserContext` and declares it as a synonym for `Context$State`.
- For each operator `op f : ... -> S` in `UserMod`, the module `EXTENDED[UserMod]` includes an overloaded constant `op [] : -> Context$S [ctor]`.
- For each constructor operator in `UserMod`, say `op f : S1 S2 ... Sn -> S [ctor]` it adds new operators

```
op f : Context$S1 S2 ... Sn -> Context$S [ctor] .
...
op f : S1 S2 ... Context$Sn -> Context$S [ctor] .
```

On substitutions

- It declares sorts `UserAssignment` and `UserSubstitution`, with the subsort relation `UserAssignment < UserSubstitution`.
- For each sort `S` in `UserMod`, it adds an operator

```
op _\_ : Qid S -> UserAssignment [ctor] .
```

- It adds operators to build substitutions:

```
op noSubst : -> UserSubstitution .
op _;_ : UserSubstitution UserSubstitution -> UserSubstitution
         [comm assoc id: noSubst] .
```

- It adds an operator

```
op _instanceOf_ : UserSubstitution UserSubstitution -> Bool .
```

  and provides equations that define the `instanceOf` relation as true when the first argument, taken as set of assignments, is a subset of the second.

On proof terms

- It declares `RuleName` as a synonym for `Qid`.
- It declares the constructor of proof terms

```
op {_|_:_} : UserContext RuleName UserSubstitution -> Trans [ctor] .
```

On transition patterns

- It declares the syntax for patterns

```
subsort Trans < TransProp .
op {_} : RuleName -> TransProp [ctor] .
op {_:_} : RuleName UserSubstitution -> TransProp [ctor] .
op {_|_} : UserContext RuleName -> TransProp [ctor] .
op top : RuleName -> TransProp [ctor] .
op top : RuleName UserSubstitution -> TransProp [ctor] .
```

- It includes equations for the relation `|=` between proof terms and patterns. Namely, rule names have to coincide; when a context appears in the pattern the one in the proof term has to be equal to it (that is, equal modulo the equations of the module); when a substitution appears in the pattern, the one in the proof term has to be an `instanceOf` it; when `top` appears in the pattern, the context in the term proof has to be empty (`[]`).

The correct way to use this `EXTENDED` operator on modules is explained in Section 3.5.

3.4.2 Performance Improvements

We have included two features aiming at improving the performance of the tool, at least in certain cases. The first is loop detection.

As our tool is designed with infinite systems in mind, loops, that is, the repetition of states, is not of great importance, at least from a theoretical point of view. However, as shown in the examples (see Section 4.4) if we want to apply it to systems with a finite number of states but with no deadlocks, loop detection is vital. Even for infinite systems, the gain in performance is sometimes large enough to turn an intractable problem into tractable.

Loop detection, however, does not come for free. It takes storage space and running time. It is not at all an easy task to find out when it pays to use loop detection. So we give the user the possibility to enable or disable it. See next section.

Note that, to detect a loop, we must not only consider the states the system visits, but also the strategy that has to be satisfied from each state. That is, it is not a loop if we visit the same state twice but with a different goal each time.

The second improvement is related to contexts. It turns out that, in practice, the use of contexts in transition patterns is not frequent. Therefore, we give the user the possibility of enabling or disabling the treatment of contexts. When it is disabled, no contexts are generated for proof terms and no comparison is done on them. According to our measurements, disabling contexts cuts running time by a 10 to 20%.

Both options are on by default.

3.5 A Brief Manual

A potential user must know, first, how to write modules, adapting the Maude specification to the syntactical requirements of the model checker and, second, what are the available commands. This is described next, beginning from the latter.

3.5.1 The Commands

There are five commands the user can issue. They must always be enclosed in parenthesis, so that it is our tool and not Maude that receives them.

Start model checking `(ismc [`d`]` s `|=` $q\ f$ `.)`

Here, d is a natural number that specifies the maximum depth in the system's state space to which the search has to be performed; s is a term of sort `State`; q is a quantifier, either the literal `exists` or `forall`; and f is a TLR* guarantee formula.

Start model checking in given module `(ismc in` m `: [`d`]` s `|=` $q\ f$ `.)`

Here, m is a string, expected to be the name of a module previously introduced to the tool. By default, when no `in` m is given, the model-checking command is executed on the most recently introduced module. So, this is the way to refer to others.

Formulas and patterns The concrete syntax for TLR* guarantee formulas is:

$$f ::= \texttt{TRUE} \mid \texttt{FALSE} \mid sp \mid \texttt{NOT}\ sp \mid tp \mid \texttt{NOT}\ tp \mid f_1\ \texttt{AND}\ f_2 \mid f_1\ \texttt{OR}\ f_2 \mid \texttt{X}\ f \mid \texttt{F}\ f \mid f_1\ \texttt{U}\ f_2$$

Here, sp is a state proposition, that is, a term of sort `StateProp`. Similarly tp is a term of sort `TransProp`. The sort `TransProp`, in particular, includes patterns, written with the following syntax:

- $p ::= \{'r\} \mid \{'r : s\} \mid \{c \mid 'r\} \mid \{c \mid 'r : s\} \mid \texttt{top}('r) \mid \texttt{top}('r,s)$
- $s ::= a \mid a\ ;\ s$
- $a ::= 'v \setminus t$

Here, r is a string expected to be the label of a rule in the Maude module, c is a term of type `UserContext`, that is, a `State` with a hole `[]`, s represents a substitution given as a set of assignments, v is a string expected to be the name of a variable in the Maude module, and t is a ground term of the same sort as v.

Results of model checking There are several possible answers to the model-checking commands:

- `Yes`, when the quantifier was `forall` and the formula was found to hold;
- `Yes` with a witness computation, when the quantifier was `exists` and the formula was found to hold;
- `No`, when the quantifier was `exists` and the formula was found not to hold;
- `No` with a counterexample, when the quantifier was `forall` and the formula was found not to hold;
- `DontKnow` followed by the number of open tasks, with any quantifier, when the search was not conclusive.

Witnesses and counterexamples can be, in particular, looping computations. In the `DontKnow` case, the tool keeps in its memory all the open tasks, so that the following command can be issued:

Deeper Model Checking `(ismc deeper [`d`] .)`
This asks the model checker to search d more levels for each open task remaining after the latest `ismc` or `deeper` command. The possible answers to this are the same explained above.

Show a task `(ismc show a task .)`
When there are open tasks, the tool shows just one of them in answer to this command.

Set options on or off There are two `set` commands:

- `(ismc set loops on .)` or `(ismc set loops off .)`
- `(ismc set contexts on .)` or `(ismc set contexts off .)`

The first instructs the tool to look (or not) for possible looping computations as it searches. It must be noted that in a loop not only states have to repeat, but also the strategies coupled with them. Detecting loops is costly, as it involves storing some information and checking for repetitions. However, in some cases it pays, even it may be the only way to reach a final answer in a system.

The command about contexts instructs the tool to use (or not) contexts, that is, generate them for proof terms and compare them with patterns. When contexts are not used in any pattern appearing in the formula to be model checked, the user gains some performance by setting contexts off.

3.5.2 The Modules

The system specification on which the model checking is to be performed has to have, at least, two modules. One of them, let us call it `UserMod`, has the whole specification of the system in the usual Maude way. Of course, it can import other modules as needed. The other module has to include the instruction `extending EXTENDED[UserMod] .` In this second module the users —having at their disposal all the infrastructure about proof terms, patterns, satisfaction. . . — can define their own atomic propositions on states or on transitions to be used in the TLR* formula. It is the name of this second module that can appear in a `ismc in` command.

Chapter 4

Examples

We include in this chapter a series of example systems. We describe each, show how they are specified in Maude, and apply our model checker to them. They can be seen both as a guide to the practical use of the model checker and as a proof of its usefulness. The complete Maude specifications for all the examples are available at http://maude.sip.ucm.es/ismc.

4.1 Faulty Channels, Attackers, and Cookies

The first two examples are the ones Meseguer proposes in [21]. The one in this section involves a fault-tolerant client-server protocol enriched with a cookie protection mechanism against denial of service (DoS) attacks. A client talks always to the same server. It sends a question and waits for an answer. The channel is faulty: messages can vanish or get duplicated. That is why a client must be ready to ask their question more than once, and a server must be ready to answer repeated questions.

Client states have five components: the client id, the id of the server they talk to, the question, the cookie, and the answer.

```
sort Client .
op [_,_,_,_,_] : ClientId ServerId Question Cookie Answer -> Client [ctor] .
```

The cookie is computed by the server and sent to the client upon request as explained below. Server states have three components: the id, a counter and a database.

```
op [_|_|_] : ServerId Nat Database -> Server [ctor] .
```

The counter is used to feed the random number generator used to get cookies. The database stores, for each client, a record with its id, its cookie (if already sent), and the answer (if already sent).

```
sorts Record Database Server .
op record : ClientId Cookie Answer -> Record [ctor] .
subsort Record < Database .
op mtDB : -> Database [ctor] .
op __ : Database Database -> Database [ctor assoc comm id: mtDB] .
```

For convenience, client and server ids, cookies, questions, and answers are all taken to be natural numbers, although cookies and answers also need *not-yet* values.

```
subsorts Nat < ClientId ServerId Question Answer Cookie .
op noAnswer : -> Answer [ctor] .
op noCookie : -> Cookie [ctor] .
```

We also need to declare the messages clients and servers are going to interchange:

```
sort Message .
op _<|_,_,_ : ServerId Cookie ClientId Question -> Message [ctor] .
op _<|_,_ : ClientId ServerId Answer -> Message [ctor] .
op _<init|_ : ServerId ClientId -> Message [ctor] .
op _<cook|_ : ClientId Cookie -> Message [ctor] .
```

The arrow `<|` marks the direction of the message, so that the addressee is always on the left.

A state is declared as a *soup* of clients, servers and messages:

```
sort State .
subsorts Client Server Message < State .
op nullState : -> State [ctor] .
op __ : State State -> State [ctor comm assoc id: nullState] .
```

Let us have a look at the rules that govern the system. The first thing a client `C` must do is telling its server `S` it is there, by sending an `init` message. The server answers storing `S` in its database, if it was not already there, and sending the cookie within a `cook` message. As it is possible that `C` was already in the database, the server looks it up and, if it finds it, reuses the same cookie.

```
rl  [init] :
    [C, S, Q, noCookie, noAnswer]
    =>
    [C, S, Q, noCookie, noAnswer] (S <init| C) .
crl [scookie] :
    (S <init| C) [S | N | D]
    =>
    if lookup(D, C) :: Record
    then [S | N | D] (C <cook| cookie(D, C))
    else [S | N + 1 | update(D, C, K, noAnswer)] (C <cook| K)
    fi
    if K := random(N) .
rl  [gcookie] :
    [C, S, Q, K, noAnswer] (C <cook| K')
    =>
    [C, S, Q, K', noAnswer] .
```

The partial function `lookup` tries to find `C` in the database `D`. The partial function `cookie` retrieves the cookie for `C` from `D`. And the function `update` does to the database what its name suggests. They are coded as follows, including the function `answer` that will be used shortly:

```
op lookup : Database ClientId ~> Record .
eq lookup(record(C, K, A) D, C) = record(C, K, A) .
op cookie : Database ClientId ~> Cookie .
eq cookie(record(C, K, A) D, C) = K .
op update : Database ClientId Cookie Answer -> Database .
eq update(record(C, K, A) D, C, K', A) = record(C, K', A) D .
eq update(D, C, K, A) = record(C, K, A) D [owise] .
op answer : Database ClientId -> Answer .
eq answer(record(C, K, A) D, C) = A .
eq answer(D, C) = noAnswer [owise] .
```

Once the client has its cookie, it can send its request. It does so by sending a message `S <| K, C, Q`, including the cookie and the question number. The server tests the cookie against its database and, if correct, sends the answer, either computing and storing it or retrieving it from the database with the function `answer`.

```
crl [req] :
    [C, S, Q, K, noAnswer]
    =>
    [C, S, Q, K, noAnswer] (S <| K, C, Q)
    if K =/= noCookie .
rl  [reply] :
    [S | N | D] (S <| K, C, Q)
    =>
    if cookie(D, C) == K
    then if answer(D, C) == noAnswer
         then [S | N | update(D, C, K, f(S, C, Q))] (C <| S, f(S, C, Q))
         else [S | N | D] (C <| S, answer(D, C))
         fi
    else [S | N | D]
    fi .
rl  [rec] :
    [C, S, Q, K, A] (C <| S, A')
    =>
    [C, S, Q, K, A'] .
```

The function `f` is the one that the server computes to find the answer to the client's request. We will have something to say about it a little later.

To model the faultiness of our system we include these rules:

```
rl [dupl] : M => M M .
rl [loss] : M => nullState .
```

The constant `nullState` was declared as the identity element for the `State` constructor, so the rule `[loss]` just removes a message from any `State`.

And all went happily until the attacker arrived:

```
sort Attacker .
subsort Attacker < State .
```

The `attacker` listens to the channel and maintains two sets of pairs: `IdIdSet`, in which it stores which server each client speaks to; and `IdCkSet`, with the cookie for each client. With this information, the attacker can send fake questions (with the wicked number `666`).

```
op attacker : IdIdSet IdCkSet -> Attacker [ctor] .
rl [learn.name] :
   attacker(IIS, ICS) (S <init| C)
   =>
   attacker(IIS idid(S, C), ICS) (S <init| C) .
rl [learn.cookie] :
   attacker(IIS, ICS) (C <cook| K)
   =>
   attacker(IIS, ICS idck(C, K)) (C <cook| K) .
rl [fake.req] :
   attacker(IIS idid(S, C), ICS idck(C, K))
   =>
   attacker(IIS idid(S, C), ICS idck(C, K)) (S <| K, C, 666) .
```

That is the system. Now we would like to check that the cookie mechanism works as a defense against the attacker. Something we surely do not want to happen is that a client has stored an answer that is not what it should be. To be more concrete, let us use this initial state:

```
op init : -> State .
eq init = [2 | 0 | mtDB]
          [1, 2, 7, noCookie, noAnswer]
          attacker(mtIdIdSet, mtIdCkSet) .
```

There is a server identified as `2`, a client identified as `1` who is going to ask `7` and an attacker with no information so far. We want to check a safety property: that when a server sends an answer and the client receives and stores it, the value stored is what it is expected to be according to the function `f`. As this is a safety property, we can only try to falsify it by model checking its negation. To this end, we define an atomic proposition on states:

```
op value-is : ClientId Answer -> StateProp [ctor] .
eq [C, S, Q, K, A] STATE |= value-is(C, A) = true .
eq STATE |= value-is(C, A) = false [owise] .
```

Here, `STATE` is a variable of sort `State`. The idea is that `value-is(C, A)` is true of a system if the client `C` is part of that system and has stored the answer `A`. The following guarantee formula represents the negation of the safety property discussed above:

```
F ( { 'reply : ('C \ 1) ; ('S \ 2) } AND
    F ( { 'rec : 'C \ 1 } AND
        X ( NOT ( value-is(1, f(2, 1, 7))))))
```

Let us ask our tool:

```
(ismc [7] init |= exists F ( { 'reply : ('C \ 1) ; ('S \ 2) } AND
                              F ( { 'rec : 'C \ 1 } AND
```

```
                                   X ( NOT ( value-is(1, f(2, 1, 7)))))) .)
```

Indeed, this can happen. Exploring to depth seven we get this witness (hiding the states):

```
... -> init -> ... -> learn.name -> ... -> scookie -> ... -> learn.cookie ->
... -> fake.req -> ... -> reply -> ... -> rec -> ...
```

So our system is rather unsafe: clients are ready to receive an answer even before having sent a request!

Some abstraction, no doubt, can be applied to this system to remove the sources of infinity, but citing Meseguer in [21]: "The point, however, is that all such efforts to obtain a tractable finite-state abstraction, and the associated theorem proving work to check confluence, coherence and preservation of state predicates for the abstraction, are not even worth it; since this simpler analysis of the system specification has already uncovered a key flaw."

Before letting this example go away, it is worth having a look at the function `f`. It is not interesting, because we don't want our result to depend on its precise definition, and this makes it quite interesting. This is the only specification for `f` we made in our system:

```
op f : ServerId ClientId Question -> Answer [ctor] .
```

Using some formalisms, other than Maude, it would have been necessary to use some fully defined example function. If that example function happened not to be injective, a false positive could show up, hiding the problem to the model checker. For us, a ground term of the form `f(s,c,q)` is only equal to itself. Of course, declaring some desired properties of `f` is possible. For instance, adding the equation

```
eq f(S, C, Q) = f(0, C, Q) .
```

ensures that all servers use the very same function.

4.2 Faulty Channels and Timing

The second example involves again a faulty channel. There are no attackers this time, but we want to be sure that a client eventually receives an answer, if it asks for it as long as needed. That is, we are interested in performing a model checking like this:

```
(ismc [11] init |= forall F { 'rec : 'C \ 1 } .)
```

Let us explain the system first.

The system evolves in phases. We assume our channel is faulty in a bounded way, so that, in each phase, only a maximum fixed number of duplications and losses are possible. A client who, in a given phase, has sent N times its question but has not received its answer, will send the same question $N+1$ times in the following phase. Thus, we expect, eventually the questions and the answers will outnumber the losses.

These are the constructors for clients, servers, and messages:

```
op [_,_,_,_,_,_] : ClientId ServerId Question Nat Bool Answer
                   -> Client [ctor] .
op [_] : ServerId -> Server [ctor] .
```

```
op _<|_,_ : ClientId ServerId Answer -> Message [ctor] .
op _<|_,_ : ServerId ClientId Question -> Message [ctor] .
```

A `Conf` is a *soup* with clients, servers, and messages. By adding a timer to it, we get a `TimedConf`, also known as a `State`.

```
sort Conf .
subsorts Client Server Message < Conf .
op nullConf : -> Conf [ctor] .
op __ : Conf Conf -> Conf [ctor comm assoc id: nullConf] .
sort TimedConf .
op {_|_} : Conf Nat -> TimedConf [ctor] .
sort State .
subsort TimedConf < State .
```

We also include two objects to account for losses and duplications:

```
sort Fault .
subsort Fault < Conf .
op dupl : Nat -> Fault [ctor] .
op loss : Nat -> Fault [ctor] .
op maxFaults : -> Nat .
eq maxFaults = 1 .
```

For simplicity we suppose only one duplication and one loss can happen in each phase. That is the meaning of `maxFaults = 1`.

Most rules are straightforward:

```
rl [reply] : [S] (S <| C, Q) => [S] (C <| S, f(S, C, Q)) .
rl [rec] : [C, S, Q, N, B, A'] (C <| S, A) => [C, S, Q, N, B, A] .
rl [dupl] : Msg dupl(s(N)) => Msg Msg dupl(N) .
rl [dupl-quit] : dupl(s(N)) => dupl(0) .
rl [loss] : Msg loss(s(N)) => loss(N) .
rl [loss-quit] : loss(s(N)) => loss(0) .
```

The two rules `[dupl-quit]` and `[loss-quit]` model the fact that not all the `maxFaults` faults must necessarily happen in each phase.

A little more interesting is this one:

```
rl [req] : [C, S, Q, N, true, noAnswer] =>
           [C, S, Q, N, false, noAnswer] (N copies S <| C, Q) .
```

The operator `copies` creates, as expected, `N` equal copies of a message. The Boolean in the fifth component of the client flags whether it is still enabled, in the present phase, to send its messages.

And this rule allows to advance to the next phase:

```
crl [tick] : {X | T} => {next(X) | s(T)} if not enabled(X) .
```

The function `enabled` tests that the system can evolve further within the present phase.

```
ops enabled enabledReq enabledReply enabledRec : Conf -> Bool .
eq enabled(dupl(D) loss(L) X) = D > 0
                                or-else L > 0
                                or-else enabledReq(X)
                                or-else enabledReply(X)
                                or-else enabledRec(X) .
eq enabledReq([C, S, Q, N, true, noAnswer] X) = true .
eq enabledReq(X) = false [owise] .
eq enabledReply((S <| C, Q) [S] X) = true .
eq enabledReply(X) = false [owise] .
eq enabledRec((C <| S, A) [C, S, Q, N, B, A'] X) = true .
eq enabledRec(X) = false [owise] .
```

The function `next` creates the initial state for the next phase, putting the possible losses and duplications to its maximum, and enabling clients to send one more message than in the previous phase:

```
op next : Conf -> Conf .
eq next(dupl(N) X) = dupl(maxFaults) next(X) .
eq next(loss(N) X) = loss(maxFaults) next(X) .
eq next([C, S, Q, N, B, A] X) =
   if A == noAnswer
   then [C, S, Q, s(N), true, A] next(X)
   else [C, S, Q, N, true, A] next(X)
   fi .
eq next([S] X) = [S] next(X) .
eq next(Msg X) = Msg next(X) .
eq next(nullConf) = nullConf .
```

We choose this initial state:

```
op init : -> State .
eq init = { [9]
            [1, 9, 7, 2, true, noAnswer]
            [2, 9, 17, 1, true, noAnswer]
            dupl(maxFaults)
            loss(maxFaults)
          | 1 } .
```

The timer is `1`, there is a server `[9]`, and two clients with ids `1` and `2`. Both clients are enabled to send their questions, respectively `7` and `17`. Client `1` will start sending two copies of its question, while client `2` will start with only one copy in the first phase.

The command

```
(ismc [11] init |= forall F { 'rec : 'C \ 1 } .)
```

answers a `Yes`, confirming what we expected.

This time, it has been a genuine guarantee formula that we have verified, while in the previous example we falsified a safety formula.

4.3 Production Rules for Grammars

Formal grammars are often presented by means of *production rules* or *string rewrite systems*. Mathematically, they are a tuple (N, S, T, R), where N is the set of nonterminal symbols, $S \in N$ is the distinguished initial symbol, T is the set of terminal symbols, disjoint from N, and R is a set of rewrite rules from strings in $(N \cup T)^*$ to other such strings. When the left-hand sides of the rules are restricted to be just elements of N, the grammars that result are context-free. This in an example taken from [16]:

$$\begin{array}{lcl} S & \to & XY \mid aSS \\ X & \to & bY \mid \epsilon \\ Y & \to & XX \end{array}$$

If we also allow for part of the context to be specified, as in this example from [27], we get more general grammars:

$$\begin{array}{lcl} S & \to & aBSc \mid abc \\ Ba & \to & aB \\ Bb & \to & bb \end{array}$$

Indeed, the language this generates is $\{a^n b^n c^n : n \in \mathbb{N}\}$, which is known not to be context-free.

Of course, rewriting logic is related to this formalism, and very well suited to model it. We need some infrastructure:

```
(mod GRAMMAR-INFRASTRUCTURE is
   sorts Terminal NonTerminal String .
   subsorts Terminal NonTerminal < String .
   op null : -> String [ctor] .
   op __ : String String -> String [ctor assoc id: null] .
   sort State .
   subsort String < State .
endm)
```

And now the examples above translate into Maude like this:

```
(mod GRAMMAR1 is
   extending GRAMMAR-INFRASTRUCTURE .
   ops a b : -> Terminal [ctor] .
   ops S X Y : -> NonTerminal [ctor] .
   rl [S1] : S => X Y .
   rl [S2] : S => a S S .
   rl [X1] : X => b Y .
   rl [X2] : X => null .
   rl [Y] : Y => X X .
endm)
```

```
(mod GRAMMAR2 is
```

```
    extending GRAMMAR-INFRASTRUCTURE .
    ops a b c : -> Terminal [ctor] .
    ops S B : -> NonTerminal [ctor] .
    rl [S1] : S => a B S c .
    rl [S2] : S => a b c .
    rl [Ba] : B a => a B .
    rl [Bb] : B b => b b .
endm)
```

These are naturally infinite systems, and asking our model checker whether a string can be generated (that is, whether it is grammatical) is quite straightforward. For instance, defining a state proposition `a3b3c3` that is only true of the string `a a a b b b c c c`, we can write:

```
(ismc in GRAMMAR2-FULL : [10] S |= exists F a3b3c3 .)
```

Our tool answers `Yes` and provides a witness in eight steps.

Another interesting question is whether the language is nonempty:

```
var St : State .
vars A B : String .
op only-terminals : -> StateProp .
eq A S B |= only-terminals = false .
eq A X B |= only-terminals = false .
eq A Y B |= only-terminals = false .
eq St |= only-terminals = true [owise] .

(ismc in GRAMMAR1-FULL : [5] S |= exists F only-terminals .)
```

The answer is `Yes`, but not in less than five steps.

We still have only a semi-decision procedure, and there are surely better tools for grammar analysis, specially for the context-free case —membership and emptiness are known to be decidable in this case. But, on the other hand, our model checker provides flexibility. For instance, we could ask whether certain string is derivable without using a given rule:

```
(ismc in GRAMMAR1-FULL : [10] S |= exists (NOT { 'X1 }) U certain-string .)
```

Or whether in fact the language of `GRAMMAR2` is the one we said:

```
(ismc in GRAMMAR2-FULL : [10] S |= exists F NOT balanced-string .)
```

Or whether a given nonterminal appears in some intermediate step of every possible derivation of terminal strings:

```
(ismc in GRAMMAR1-FULL : [10] S |= forall (NOT hasX) U only-terminals .)
```

4.4 A Finite System: MUTEX

Just in order to show that our tool works nicely on finite systems too, let us consider this example taken from [10], that specifies a simple mutual-exclusion algorithm based on tokens:

```
(mod MUTEX is
   ...
   rl [a-enter] : $ [a, waiting] => [a, critical] .
   rl [b-enter] : * [b, waiting] => [b, critical] .
   rl [a-exit]  : [a, critical] => [a, waiting] $ .
   rl [b-exit]  : [b, critical] => [b, waiting] * .
endm)
```

We have only included the rules, that are pretty self-explanatory. The system is not only finite and small, but also deterministic. That should be an easy bite.

Before invoking the model checker, we ensure loop detection is turned on:

```
(ismc set loops on .)
```

The two natural questions to ask are whether both processes can enter their critical section, and whether they can do it at the same time.

```
(ismc [3] [a,waiting] [b,waiting] $ |= forall (F crit(a)) AND (F crit(b)) .)
(ismc [2] [a,waiting] [b,waiting] $ |= exists F (crit(a) AND crit(b)) .)
```

We expect a `Yes` and a `No`, respectively. As it happens, the second question results in the expected `No`, but finding a loop in just two steps, which is quite surprising. The definitive proof that something is going wrong is that the first command produces a negative answer with the following looping computation:

```
[a,waiting][b,waiting]$
-> a-enter ->
[a,critical][b,waiting]
-> a-exit ->
$[a,waiting][b,waiting]
-> a-enter ->
[a,critical][b,waiting]
```

Only process `a` is entering its critical section. Of course, there is an error in the specification, as process `a` is leaving the token `$`, that only allows itself to enter again. By the way, note that in the above four-step loop, the states are repeating in shorter loops, but the associated strategies (not shown) are not: at start we are looking for paths that satisfy both `crit(a)` and `crit(b)`. After we find a state that satisfies `crit(a)`, our goal changes.

The error is easy to fix:

```
(mod MUTEX is
   ...
   rl [a-enter] : $ [a, waiting] => [a, critical] .
   rl [b-enter] : * [b, waiting] => [b, critical] .
   rl [a-exit]  : [a, critical] => [a, waiting] * .
   rl [b-exit]  : [b, critical] => [b, waiting] $ .
endm)
```

Now, the question about both processes eventually entering their critical sections is answered with a `Yes`, even with loop detection disabled (searching up to depth three). And the

question about mutual exclusion produces a `No` (searching up to depth four) only when loop detection is on; otherwise it is never able to give a definitive answer.

In this last case, if we ask the model checker at each step about its (unique) pending task, we get this sequence of pairs state / strategy:

```
[a,waiting][b,waiting]$
any ; any * ; idle . crit(a) And crit(b)

[a,critical][b,waiting]
any ; any * ; idle . crit(a) And crit(b)

*[a,waiting][b,waiting]
any ; any * ; idle . crit(a) And crit(b)

[a,waiting][b,critical]
any ; any * ; idle . crit(a) And crit(b)
```

The strategy happens to be always the same, which makes sense: We are always looking for a possible future violation of mutual exclusion, and this time `crit(a)` and `crit(b)` have to be satisfied simultaneously. Thus, when the state repeats, a loop is found.

Chapter 5

A Case Study: The MSI Cache Coherence Protocol

For this chapter we wanted to find a problem large enough, interesting enough and, if possible, not previously modeled in rewriting logic. Our aim was to test the usefulness of rewriting logic, Maude and, above all, our new tool both in the modeling —as a debugging tool, say— and in proving the desired properties of the system. We chose cache coherence protocols, that seem to fulfill all the requirements. In particular, we focus attention on the MSI protocol. Note that one of the requirements was not that the problem was amenable to any particular kind of model checking or modeling: this is what we wanted to test. We remind the reader that the complete Maude specifications are available at `http://maude.sip.ucm.es/ismc`.

5.1 The Setting

Let us describe the problem. In modern computers, typical processors outperform typical memories in speed. Accesses to memory, either to read or write data, are a bottleneck. There would be no point in expending money on a better processor, because the memory hinders the performance of the computer. There exist better memory technologies, that produce faster memories, but they are notably more expensive. Here is where the notion of *cache memory* proves appropriate.

The idea is to use a large, slow main memory to store all the data, and one or more small, fast memories to store copies of the frequently or currently used parts. These small memories are called cache memories or just *caches*. Different computer architectures use them in different ways. Typically, when the processor needs some data, it asks its cache for it. If the cache has not the data, it fetches it from main memory, then stores the data and its address, and passes it to the processor. Next time the processor needs to work with that piece of data, it is already available in the cache. There is a piece of hardware, called the *bus*, whose task is to put in contact caches with the main memory (and, to some extent, also each cache with each other, as we will see later). Writes are also done to the cache.

Here a problem shows up: At any given moment, several caches can be storing a copy of the same part of the main memory. If one of the processors modifies it on its cache, the others would be left with an invalid data. This is the *cache coherence* problem. To solve the problem, so-called *cache coherence protocols* have been devised. In this section we consider a simple and well-known one: the MSI protocol, described, for instance, in [14]. We will explain it, model it at several levels of abstraction, and verify some of its coherence properties.

Many different possibilities exist in computer architecture, and different architectures call for different protocols. So, before describing how the MSI protocol works, let us fix the characteristics of the ideal computer on which it is designed to work (see Figure 5.1). To begin with, our computer has one main memory, one bus, and an undefined number of processors and caches. A processor uses only a cache and a cache serves only a processor. (Some computers include caches that can be shared by several processors, but the MSI protocol is not useful for them.) A processor can only access memory through its cache. (Some architectures allow processors direct access to main memory in some circumstances.) The processors have access to all memory addresses, that is, they are *tightly coupled*.

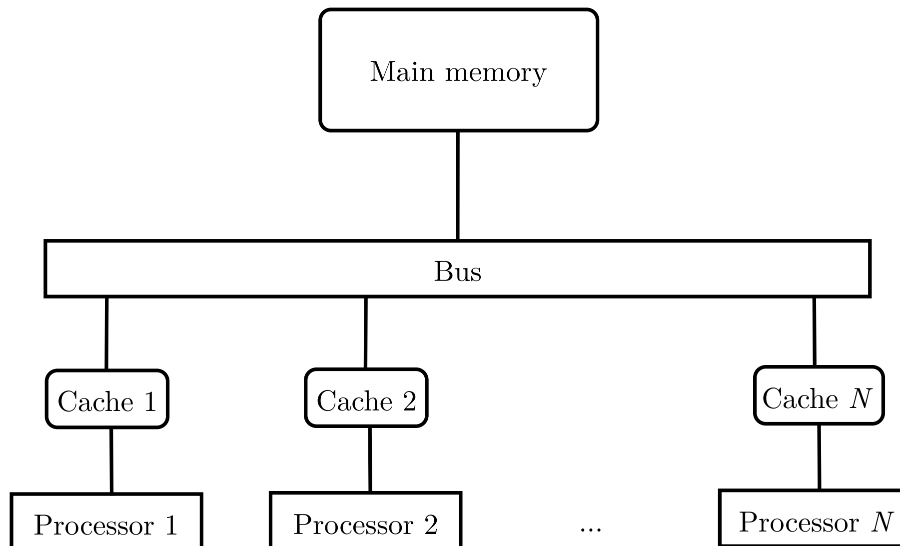

Figure 5.1: The structure of our computer

The bus does not allow direct communication of data from cache to cache, but only through the main memory. However, the caches are able to *snoop* the bus, that is, to monitor it to detect when another cache is trying to access a main-memory address that it has replicated. As we will see, this is a necessary condition for MSI. Beware that, through snooping, a cache cannot see the data being read or written by another cache, but only its address in main memory.

Architectures include *policies*, rules on the workings of caches. One such policy is the choice of the size for the smallest chunk of memory a cache is able to read or write. This

smallest chunk is called a *cache line*. The size of the line does not concern us, because the protocol works for all of them. More interesting is the writing policy. Some architectures use a *write-through* policy. This means that each time data is written on a cache, it is also written, through the bus, to main memory. This is a simple but costly policy. Rather, our ideal computer is using a *copy-back* policy. This means that data written by the processor to the cache is not immediately written to main memory, but only when circumstances make it necessary.

One last point to consider is that operations on the bus are taken to be atomic or, equivalently, that a command put on the bus (like an order to write to main memory, for instance) is fully dealt with by the bus, the main memory and all the caches before they do anything else. Reality is usually not that simple, but considering technicalities on the bus side would make the model unnecessarily complex.

5.2 The MSI Protocol

In the MSI protocol, like in many others, each cache line is marked with flags that determine the validity of the information they store. Thus, in MSI a line can be in one of three modes:

invalid: after being read from main memory, the line has been modified in another cache;

shared: the line is valid, and has not been modified in this or another cache, so that every copy of the line stored in any cache is valid;

modified: (also known as *dirty*) the line has been modified in this cache one or more times, so that this is the only cache to store a valid copy; even the copy in main memory is invalid.

The initials of these three states (bottom up, as we have listed them) give its name to the protocol. Throughout this exposition, we abbreviate the three modes as `inv`, `shr`, and `mdf`.

The protocol determines the actions that a cache must perform according to the orders that arrive to it. It is described below. Eight kinds of orders need to be considered. The first four deal with orders the cache receives from its processor, either reads or writes. The last four are orders initiated by another processor, that the cache snoops from the bus. In each case, a *hit* happens when the order the cache receives refers to a memory address that the cache has already stored. The opposite to a hit is a *miss*.

All through this exposition we assume that each cache contains a unique line of information. This is an unrealistic but sensible simplification: Each time a hit happens in a cache, it happens to a particular line, and all others are irrelevant to that operation. If it is a miss what happens, it is, by definition, a miss for all lines, but it is just one of them that will need to be cleared in order to house the new data, so that, again, the other lines keep untouched. (By the way, the choice of the line to be evicted is by itself an important problem, but our protocol works for any way of choosing.) Therefore, sometimes we talk about caches or lines interchangeably, and when we say that a cache is in a certain mode, it must be understood that its only line is.

The following has been adapted mainly from [17] (indeed, it is the "N+1" protocol what is discussed there, with some unimportant differences):

Processor read miss:

- If in mode mdf, then evict line to main memory.
- Read requested line from the bus and store it.
- Send data to the processor.
- Change mode to shr.

Processor read hit:

- Send data to the processor.
- No mode change.
- (Does not occur while in mode inv.)

Processor write miss:

- If in mode mdf, then evict line to main memory.
- Send "invalidate" signal to the bus, so that other caches can snoop it.
- Write to cache line.
- Change mode to mdf.

Processor write hit:

- If in mode shr, then send "invalidate" signal to the bus, so that other caches can snoop it.
- Write to cache line.
- Change mode to mdf.
- (Does not occur while in mode inv.)

Bus miss (read, write, or invalidate):

- No response.
- No mode change.

Bus invalidate hit:

- Change mode to inv.

Bus read hit:

- If in mode mdf, then:
  - abort snooped read,

    - write modified line to main memory,
    - send *retry* signal through the bus.
- Change mode to `shr`.
- (Does not occur while in mode `inv`.)

Bus write hit:

- (Does not occur.)

A few explanations may be in order. A specially simple case is the "bus miss." On snooping this, a cache would think: "Someone is reading from or writing to an address I don't have stored, so I have nothing to do." For a slightly more complex case, consider the "processor write hit." On detecting this, a cache in mode `shr` would think: "My processor needs to write to a line I already have stored. I will just write the new data. But this is the first time I modify this line, so I will ask the bus to send an *invalidate* signal, so that other caches are aware that some change has happened. And I will change to `mdf`."

The first action for cases "processor read miss" and "processor write miss" needs explanation. In these cases the processor is trying to access an address that is not present in its cache. For this to be possible, some line present in the cache must be cleared and its space reused. If the line chosen to be cleared is in mode `mdf`, that is, it is the only valid copy, the line must be copied (evicted) to main memory, and only then the new data can be brought to the cache, either from main memory —if the order was a read— or from the processor —if the order was a write.

The most complex case is the "bus read hit" when it is snooped by a cache in mode `mdf`. Let the cache speak: "Someone is trying to read from main memory a line whose only valid copy is the one I have stored. I cannot let this happen. This is what I will do: I will abort the read, then I will copy my line to main memory, and then I will send a *retry* signal so that the cache that was trying to read knows it must try again. Finally, I will change to `shr`." Of course, the architecture of the bus and of the caches makes all this possible.

Finally, why can't a bus write hit happen at all? All actions that can take a line to `mdf` also send an "invalidate" signal. Thus, if a line is in `mdf`, no other line stores that address. The only bus writes in the MSI protocol are evictions of lines that are in mode `mdf`, thus, no hit can happen elsewhere.

5.3 Level 1 Model

The first Maude model we present is basically a translation of the description above, with a few simplifications. Let us begin having a look at the data structures we use.

A `Line` consists of an address and the data stored:

```
sort Line .
op line : Address Data -> Line [ctor] .
```

Both `Address` and `Data` are natural numbers, for simplicity. The sort `Mode` is straightforward:

```
sort Mode .
ops mdf shr inv : -> Mode [ctor] .
```

We define a sort `ChipId` (as a `Nat`) whose aim is to be the nexus between a processor and its cache. So we think of a chip as containing a processor-cache couple. A CPU, or processor, contains just its `ChipId` and a Boolean that indicates whether it sent a message to its cache and is waiting for the answer. A cache registers its `ChipId`, its mode and its line of information:

```
sort CPU .
op cpu : ChipId Bool -> CPU [ctor] .
sort Cache .
op cache : ChipId Mode Line -> Cache [ctor] .
```

As already explained above, a cache only contains one line.

The main memory is declared as a set of lines enclosed in double curly brackets:

```
sort MemoryContents .
subsort Line < MemoryContents .
op mtMemoryContents : -> MemoryContents [ctor] .
op __ : MemoryContents MemoryContents -> MemoryContents
        [ctor comm assoc id: mtMemoryContents] .
eq MC:MemoryContents MC:MemoryContents = MC:MemoryContents .
sort Memory .
op {{_}} : MemoryContents -> Memory [ctor] .
```

The bus is not a distinct entity in our implementation. There are just `BusMessages` loose in the state. There are also `LocalMessages`, that is, messages between a processor and its cache, whose means of transmission is of no concern to us either.

```
sorts BusMessage LocalMessage .
op bus-read : ChipId Address -> BusMessage [ctor] .
op bus-hereur : ChipId Line -> BusMessage [ctor] .
op read : ChipId Address -> LocalMessage [ctor] .
op hereur : ChipId Line -> LocalMessage [ctor] .
op write : ChipId Line -> LocalMessage [ctor] .
```

Some functions described below are also related to the tasks of the bus.

We want to control the amount of memory addresses and of possible data values available, so that they can be kept to the minimum we need in each moment. We do so by defining a sort of sets on natural numbers, `NatSet`, and these two sorts:

```
sorts AddressRange DataRange .
op aRange : NatSet -> AddressRange [ctor] .
op dRange : NatSet -> DataRange [ctor] .
```

We will shortly see its use.

A `State` is defined as a *soup* of the described elements, enclosed in single curly brackets:

```
sort StateContents .
```

```
subsorts CPU Cache Memory BusMessage LocalMessage
         AddressRange DataRange < StateContents .
op mtStateContents : -> StateContents [ctor] .
op __ : StateContents StateContents -> StateContents
        [ctor comm assoc id: mtStateContents] .
sort State .
op {_} : StateContents -> State [ctor] .
```

Some provisos are missing that are important. For instance, only a `Memory`, an `AddressRange` and a `DataRange` can exist and there should be as many `Cache`s as `CPU`s, coupled by id. These and others have to be enforced in the initial states we use and in the rules that govern the system.

Let us consider the first rule:

```
crl [read] :
    { cpu(Id, false) aRange(N ; NS) SC }
    =>
    { cpu(Id, true) read(Id, N) aRange(N ; NS) SC }
    if not hasBusMsg(SC) .
```

The variable `SC` has sort `StateContents` and captures the part of the state about which we do not need to be precise. This rule allows processors to start a reading by their own initiative. Here, the `AddressRange` is used. The set of available addresses is of the form `N ; NS`, that is, a natural number `N` and a set of other numbers `NS`. It is `N` that is used as address to be read: `read(Id, N)`. As the operator `;` is commutative and associative, `N` can match any of the available addresses. So, an instance of this rule can be executed, nondeterministically, for each address. The same trick is used to start a writing, this time to choose both an address and a data value:

```
crl [write] :
    { cpu(Id, false) aRange(N ; NS) dRange(N' ; NS') SC }
    =>
    { cpu(Id, true) write(Id, line(N, N'))
      aRange(N ; NS) dRange(N' ; NS') SC }
    if not hasBusMsg(SC) .
```

A word about the conditions in these rules. We want bus operations to be considered as atomic. Thus, when a bus message is present, it must have priority over anything else. That is why these and other rules are only enabled when there is no bus message pending. Besides, its seems a nice assumption that no new command to the cache can be issued until the previous one, either read or write, has been thoroughly processed. That is why a `CPU` can issue a new command to its cache only when it is not waiting for the answer to a previous one, that is, when its Boolean is `false`.

Another simple rule:

```
crl [read-hit] :
    { cache(Id, Md, line(A, D)) read(Id, A) SC }
    =>
```

```
    { cache(Id, Md, line(A, D)) hereur(Id, line(A, D)) SC }
    if not hasBusMsg(SC) .
```

The processor is trying to read the line stored in the cache, so the answer is immediate. There is also a rule by which the processor receives the data. Indeed, it just throws it away, as we are not interested in the processor's inner workings:

```
rl [read-done] :
   { cpu(Id, true) hereur(Id, L) SC }
   =>
   { cpu(Id, false) SC } .
```

Something a little bit more involved:

```
crl [write-hit] :
    { cpu(Id, true) cache(Id, Md, line(A, D)) write(Id, line(A, D'))) SC }
    =>
    { cpu(Id, false) cache(Id, mdf, line(A, D'))
      (if Md == shr then invalidate(Id, A, SC) else SC fi) }
    if not hasBusMsg(SC) .
```

This is a write to the address the cache has already stored. But in case the address was also stored in some other caches, that caches' contents become invalid. This task is done, in real computers, by the bus and the snooping caches. In our model, we use the function `invalidate`, defined like this:

```
op invalidate : ChipId Address StateContents -> StateContents .
ceq invalidate(Id, A, cache(Id', Md, line(A, D)) SC) =
    cache(Id', inv, line(0, 0)) invalidate(Id, A, SC)
   if Id =/= Id' .
eq invalidate (Id, A, SC) = SC [owise] .
```

Now, a write miss:

```
crl [write-miss] :
    { MM cpu(Id, true) cache(Id, Md, line(A, D)) write(Id, line(A', D')) SC }
    =>
    { (if Md == mdf then update(MM, line(A, D)) else MM fi)
      cpu(Id, false) cache(Id, mdf, line(A', D')) invalidate(Id, A', SC) }
    if A =/= A' /\ not hasBusMsg(SC) .
```

The `update` function updates the memory with the given line. It corresponds to the "evict line to main memory" we used in Section 5.2. As we are locally modifying address `A'`, we need to invalidate other caches as in rule `write-hit`.

A read miss is treated like this:

```
crl [read-miss] :
    { MM cache(Id, Md, line(A, D)) read(Id, A') SC }
    =>
    { (if Md == mdf then update(MM, line(A, D)) else MM fi)
```

```
    cache(Id, Md, line(A, D)) bus-read(Id, A') SC }
  if A =/= A' /\ not hasBusMsg(SC) .
```

The eviction is present again. Also, as the requested address `A'` is not in the cache, we need to ask the bus for it. Several reactions to this bus process are possible, and we found it clearer to code it by a bus message `bus-read` than by a function. There are two rules that deal with this message. This is the first one:

```
crl [bus-read-miss] :
    { MM bus-read(Id, A) SC }
    =>
    { MM bus-hereur(Id, lookup(MM, A)) SC }
    if not hasHit(Id, A, SC) .
```

It is only enabled when `hasHit` is false, that is, when no other cache apart from `Id` has address `A` stored. In this case, the value is retrieved from main memory using the `lookup` function.

When there is a hit, this is what happens:

```
crl [bus-read-hit] :
    { MM cache(Id', mdf, line(A, D)) bus-read(Id, A) SC }
    =>
    { update(MM, line(A, D))
      cache(Id', shr, line(A, D)) bus-hereur(Id, line(A, D))
      SC }
    if Id =/= Id' .
```

First thing to note here is that we are only interested in hits to lines in mode mdf (indeed, the `hasHit` function only considers them): a hit to an inv line is impossible (both in practice and in our implementation, as lines are cleared to `line(0, 0)` when they are invalidated), and a cache in mode shr has stored the same contents as the main memory, so the first rule is valid.

In this rule we are not following literally the protocol, but are simplifying it. The protocol's description says the reading must be aborted, the main memory updated with the latest value, and a *retry* signal sent. All that is necessary, because the buses for which the MSI protocol was devised do not allow the direct interchange of data "cache to cache." In our model, however, we are letting the hit cache put the answer directly on the bus (updating the main memory as well).

To finish, here is how the cache receives a `bus-hereur` message and informs the processor:

```
rl [bus-read-done] :
   { cache(Id, Md, L) bus-hereur(Id, L') SC }
   =>
   { cache(Id, shr, L') hereur(Id, L') SC } .
```

That is the system model. It is a large model, with many rules, but a good portion of it is deterministic, because of the priority we give to bus messages. We want to check coherence. There are several ways to put that in words, or in temporal formulas. Also, this

is a finite model, so that Maude's LTL model checker can be used. Indeed, consider for example the state proposition we will call `shr-agree`: it holds for states in which all caches in mode `shr` that store the same memory address agree in the value stored, and also agree with the value stored in the main memory. For our tool, it could be defined like this:

```
op shr-agree : -> StateProp [ctor] .
ceq { cache(Id, shr, line(A, D)) cache(Id', shr, line(A, D')) SC }
     |= shr-agree
     = false
   if D =/= D' .
ceq { cache(Id, shr, line(A, D)) {{line(A, D')}} SC } |= shr-agree
     = false
   if D =/= D' .
eq { SC } |= shr-agree = true [owise] .
```

We surely want this property to be true in every possible future state. So we ask Maude's LTL model checker:

```
red modelCheck(init, [] shr-agree) .
```

from the initial state

```
op init : -> State .
eq init = { cpu(1, false) cache(1, mdf, line(1, 2))
            cpu(2, false) cache(2, inv, line(0, 0))
            {{line(1, 1)}} aRange(1 ; 2) dRange(1 ; 3) } .
```

Maude quickly answers affirmatively. This question is not possible to express in the tool we are presenting, because `[] shr-agree` is not a guarantee formula.

Two only caches and two only memory addresses can be seen as a too simplistic model. However, as Pong and Dubois wrote in [23] "several studies have demonstrated that most design errors can be found quickly in small-scale models, which suggests that this method [of state enumeration] is a useful debugging tool in the early design phase."

Let us do next something a little different: we wonder whether invalidating is really needed. Namely: if we remove invalidation from our system, does `[] shr-agree` still holds? Removing invalidation in our system can be easily done by commenting out one of the equations for function `invalidate`, so that it becomes:

```
op invalidate : ChipId Address StateContents -> StateContents .
eq invalidate (Id, A, SC) = SC .
```

We could again ask the LTL model checker. But this time we can also ask our brand new model checker for the satisfiability of the negation:

```
(ismc [8] init |= exists F NOT shr-agree .)
```

We get a `Yes` and this witness (edited to show only the rules and the final state):

```
init
-> read -> ... -> read-miss -> ... -> bus-read-hit -> ...
-> bus-read-done -> ... -> write -> ... -> write-hit -> ...
```

```
-> read -> ... -> read-miss ->
{ {{line(1, 1)}} aRange(1 ; 2) dRange(1 ; 3)
  cpu(1, true) cpu(2, true) cache(1, mdf, line(1, 1))
  cache(2, shr, line(1, 2)) bus-read(1, 2) hereur(2, line(1, 2)) }
```

So, yes, invalidating is necessary, and we restore it to go on.

Our logic is richer than LTL, because it includes transition properties. Taking profit of it, we can think about a behavioral approach to check the protocol. What we can do is setting up an initial state with an appropriate configuration, including also a `read` on the state, and check that, eventually, the processor receives the correct result. To be concrete, this is our new initial state:

```
op init2 : -> State .
eq init2 = { cpu(1, false) cache(1, mdf, line(1, 3))
             cpu(2, true) cache(2, inv, line(0, 0)) read(2, 1)
             {{line(1, 1)}} aRange(1) dRange(4) } .
```

Processor `2` wants to read the contents of memory address `1`. That information is stored in the main memory, but it is cache `1` who has the latest value. We want to check that, eventually, cache `2` receives `line(1, 3)`. With `aRange(1)` and `dRange(4)`, we have included the possibility for processors to initiate new reads or writes to address `1` with a different value `4`; this is not a big range of possibilities, but it is all we need to try to interfere with the reading. We would like to model check with a command like this:

```
(ismc [10] init2 |=
           forall F { 'read-done : ('Id \ 2) ; ('L \ line(1, 3)) } .)
```

Unfortunately, even the more general

```
(ismc [4] init2 |= forall F { 'read-done } .)
```

produces a `No` after having found this looping computation (slightly edited for clarity):

```
init2
-> write ->
{ {{line(1, 1)}} aRange(1) dRange(4)
  cpu(1, true) cpu(2, true) cache(1, mdf, line(1, 3))
  cache(2, inv, line(0, 0)) read(2, 1) write(1, line(1, 4)) }
-> write-hit ->
{ {{line(1, 1)}} aRange(1) dRange(4)
  cpu(1, false) cpu(2, true) cache(1, mdf, line(1, 4))
  cache(2, inv, line(0, 0)) read(2,1) }
-> write ->
{ {{line(1, 1)}} aRange(1) dRange(4)
  cpu(1, true) cpu(2, true) cache(1, mdf, line(1, 4))
  cache(2, inv, line(0, 0)) read(2, 1) write(1, line(1, 4)) }
-> write-hit ->
{ {{line(1, 1)}} aRange(1) dRange(4)
  cpu(1, false) cpu(2, true) cache(1, mdf, line(1, 4))
  cache(2, inv, line(0, 0)) read(2, 1) }
```

In it, the last state is the same as the middle one (the accompanying strategy is not shown, but it is the same as well). The problem is clearly unfairness: the system is only paying attention to processor `1`.

For a small dose of reassurance, we try this:

```
(ismc [4] init2 |=
          exists F { 'read-done : ('Id \ 2) ; ('L \ line(1, 3)) } .)
```

The answer is a `Yes`: that is, *some* computation gives the correct result to processor `1`. Not enough, certainly.

There are several possibilities to go on. The perfect one would be to add a fairness condition to our formula, something such as

$$(\mathbf{F}\,\mathbf{G}(\text{message } M \text{ is available}) \rightarrow \mathbf{G}\,\mathbf{F}(\text{message } M \text{ is dealt with})) \rightarrow \text{our formula}$$

(that is, if it holds that no message is always ignored if it is always available, then our formula holds as well). But typical fairness conditions are not guarantee formulas. We could instead add fairness within the system specification, but modifying a system seems not the best way to check it.

Somewhat more satisfying can be this:

```
op init3 : -> State .
eq init3 = { cpu(1, true) cache(1, mdf, line(1, 3)) read(1, 2)
             cpu(2, true) cache(2, mdf, line(2, 3)) read(2, 1)
             {{line(1, 1) line(2, 1)}} aRange(1) dRange(4) } .
```

This is pretty much like `init2`, but this time we have included a read message for each processor, to which the other cache has the correct value. One or the other processor has to receive its correct answer:

```
(ismc [7] init3 |=
          forall F ( { 'read-done : ('Id \ 1) ; ('L \ line(2, 3)) } OR
                     { 'read-done : ('Id \ 2) ; ('L \ line(1, 3)) } ) .)
```

And indeed the answer is now `Yes`.

This has been the most complex and detailed of our models. We look now for some abstraction.

5.4 Level 2 Model

Surprisingly enough, experts on the field realized that, in order to check cache coherence, data is not the important thing to focus on. In general, hardware is reliable in performing the changes in cache lines and in line modes consistently, in the way described in Section 5.2. For instance, every time a "processor read miss" happens, the mode changes to shr. Taking this consistency for granted, only modes matter. For instance, a property we want to hold is that at most one cache is in mode mdf at a given time. If this does not hold, something bad has surely happened. The point, again, is that these kinds of properties on modes are

enough to ascertain the validity of a protocol, if we trust the bus architecture is doing the right movements of data.

According to these observations, our level 2 model does not include any actual data. In particular, no main memory is needed. The following table is an extract from the description of the MSI protocol in Section 5.2, showing only mode changes. The symbol $\times$ means that this case does not occur, and the symbol $=$ means that there is no mode change. Sometimes, when there is no mode change, I have preferred to repeat the mode's name instead of using $=$, because it makes the table more uniform *row-wise*, and more ready for implementation.

	inv	**shr**	**mdf**
Processor read miss	shr	shr	shr
Processor read hit	$\times$	$=$	$=$
Processor write miss	mdf	mdf	mdf
Processor write hit	$\times$	mdf	mdf
Bus miss	$=$	$=$	$=$
Bus invalidate hit	$\times$	inv	inv
Bus read hit	$\times$	shr	shr
Bus write hit	$\times$	$\times$	$\times$

The table shows that the only bus operations that we must care about are "invalidate hit" and "read hit." The others either are impossible or do not imply changes at all. Thus, we complement the table above with the following one, that shows which bus operations must be considered in each case upon receiving each processor request. A dash – means nothing must be done through the bus.

	inv	**shr**	**mdf**
Processor read miss	read	read	read
Processor read hit	–	–	–
Processor write miss	invalidate	invalidate	invalidate
Processor write hit	–	invalidate	–

Although in this model we only care about modes, we also have to store memory addresses, because it is address coincidence what determines which caches have to react to which bus messages. Another simplification that we allow ourselves is the absence of processors in the model. It is the caches that have the initiative to start a read or write. Or, if you prefer, we consider the processor and the cache to be integrated in a single object which, for convenience, we just call *cache*.

With this in mind, this is the new definition of a `Cache`:

```
sort Cache .
op cache : CacheId Mode Address -> Cache [ctor] .
```

and this is a `State`:

```
sort StateContents .
subsorts Cache AddressRange < StateContents .
op mtStateContents : -> StateContents [ctor] .
op __ : StateContents StateContents -> StateContents
        [ctor comm assoc id: mtStateContents] .
sort State .
op {_} : StateContents -> State [ctor] .
```

We still need to use `AddressRange`, but not `DataRange`.

We only need three rules and two functions. For a read miss:

```
crl [read-miss] :
    { cache(Id, Md, A) aRange(N ; NS) SC }
    =>
    { cache(Id, shr, N) aRange(N ; NS) map-bus-read(Id, N, SC) }
    if A =/= N .
```

In words: cache `Id` is storing address `A` and starts a reading to a different address `N` (it must be different, as it is a miss). It changes its mode to shr and tells all other caches about its change. The function `map-bus-read` changes to mdf the mode of every cache storing the newly read address:

```
op map-bus-read : CacheId Address StateContents -> StateContents .
eq map-bus-read(Id, A, cache(Id', mdf, A) SC) =
   cache(Id', shr, A) map-bus-read(Id, A, SC) .
eq map-bus-read(Id, A, SC) = SC [owise] .
```

Note that we are always supposing cache ids are unique, so that there is no need to check whether `Id =/= Id'` in the first equation.

A "write miss" includes invalidations, as already explained:

```
crl [write-miss] :
    { cache(Id, Md, A) aRange(N ; NS) SC }
    =>
    { cache(Id, mdf, N) aRange(N ; NS) map-invalidate(Id, N, SC) }
    if A =/= N .
```

The function `map-invalidate` works in a similar way to `map-bus-read`:

```
op map-invalidate : CacheId Address StateContents -> StateContents .
eq map-invalidate(Id, A, cache(Id', Md, A) SC) =
   cache(Id', inv, 0) map-invalidate(Id, A, SC) .
eq map-invalidate(Id, A, SC) = SC [owise] .
```

The case "write hit" is impossible for mode inv and nothing needs to be done for mdf, so only mode shr is included.

```
rl [write-hit] :
   { cache(Id, shr, A) SC }
   =>
   { cache(Id, mdf, A) map-invalidate(Id, A, SC) } .
```

Also, this case is a hit, so the stored address do not change.

The code for this model is quite smaller than the previous one, but it is less deterministic. Indeed, the state space is finite, but large enough to cause difficulties to our model checker, if it is a decision algorithm (not semi-decision) what we are looking for.

The key property is this: for each address `A`, if a cache storing `A` is in mode mdf, then no other cache storing `A` can be in modes mdf or shr. We can code this per address or for all addresses at once:

```
op coherent : Address -> StateProp [ctor] .
eq { cache(Id, mdf, A) cache(Id', mdf, A) SC } |= coherent(A) = false .
eq { cache(Id, mdf, A) cache(Id', shr, A) SC } |= coherent(A) = false .
eq St |= coherent(A) = true [owise] .
op coherent : -> StateProp [ctor] .
eq { cache(Id, mdf, A) cache(Id', mdf, A) SC } |= coherent = false .
eq { cache(Id, mdf, A) cache(Id', shr, A) SC } |= coherent = false .
eq St |= coherent = true [owise] .
```

The author must confess he needed the help of the model checker to find and fix a couple of errors in the above implementation. After that necessary fixing, a command like the following finds no incoherent states:

```
(ismc [8] init |= exists F NOT coherent .)
```

This is a possible initial state to use:

```
eq init = { cache(1, inv, 0) cache(2, inv, 0) aRange(1 ; 2) } .
```

5.5 Level 3 Model

The last model is the most abstract one and —finally!— an infinite one. Two properties of the previous model must be noted: First, it is enough to establish coherence per address. If coherence holds for the set of caches storing a given arbitrary address, it holds for all of them. Second, all caches are treated equal, and only the amount of them in each mode matters for coherence. For instance, rephrasing what we said a few lines above: for each address, $\#\mathsf{mdf} \geq 1 \Rightarrow \#\mathsf{mdf} = 1$ and $\#\mathsf{shr} = 0$.

Now, we choose a fixed address and try to establish coherence for it. In the rest of this chapter, address `A` is a fixed but arbitrary address. The question we want to study now is: How does the amount of caches storing `A` in each of the modes vary when performing reads and writes? Let us introduce a little bit of notation: $M_{\mathtt{A}}$ is the number of caches in mode mdf that store the fixed address `A`; in the same way, $S_{\mathtt{A}}$ is the number of caches in mode shr that store `A`; and the same for $I_{\mathtt{A}}$. Or, wait: $I_{\mathtt{A}}$ should count the inv caches that store `A`. But this is absurd, because an inv cache, by definition, does not store any valid address at all. Therefore, we want to study how the pair $(S_{\mathtt{A}}, M_{\mathtt{A}})$ varies as the system evolves.

Once again we have to play the not-always-funny game of case distinction. The author feels his duty to list all the cases; the reader can decide by himself what to do. There are

the four usual cases: (read | write) (hit | miss), with their variants according to the previous mode of the cache. But now, in the "miss" cases, we also have to distinguish whether we have a miss to a cache storing A that will cease to store A, or to a cache not storing A that will subsequently store it.

- **Read miss to an inv cache**: If it is an inv cache what we are trying to read, it will become a shr cache storing A, so $S_{\mathtt{A}}$ will increase by one. In addition, if there already was a mdf cache storing A, it will share its data and become shr, so that $M_{\mathtt{A}}$ will decrease by one and $S_{\mathtt{A}}$ increase. This gives two possible transitions:

$$(S_{\mathtt{A}}, 0) \to (S_{\mathtt{A}} + 1, 0) \qquad (S_{\mathtt{A}}, M_{\mathtt{A}}) \to (S_{\mathtt{A}} + 2, M_{\mathtt{A}} - 1) \text{ if } M_{\mathtt{A}} > 0.$$

- **Read miss to a shr cache storing A**: The contents of the cache, its address and its data, are going to be replaced, so that $S_{\mathtt{A}}$ will decrease by one:

$$(S_{\mathtt{A}}, M_{\mathtt{A}}) \to (S_{\mathtt{A}} - 1, M_{\mathtt{A}}) \text{ if } S_{\mathtt{A}} > 0.$$

- **Read miss to a shr cache that will begin to store A**: The cache will become a new element to count in $S_{\mathtt{A}}$. Also, as in a previous item, if there already was a mdf cache storing A, it will share its data and become shr, so that $M_{\mathtt{A}}$ will decrease by one and $S_{\mathtt{A}}$ increase. This gives two possible transitions, that happen to be the same we got for a read miss to an inv cache.

- **Read miss to a mdf cache storing A**: The address and its data are going to be replaced (after been evicted to main memory), so that $M_{\mathtt{A}}$ will decrease by one:

$$(S_{\mathtt{A}}, M_{\mathtt{A}}) \to (S_{\mathtt{A}}, M_{\mathtt{A}} - 1) \text{ if } M_{\mathtt{A}} > 0.$$

- **Read miss to a mdf cache that will begin to store A**: The cache will become a new element to count on $S_{\mathtt{A}}$. We get again the same two transitions displayed in the case of a read miss to an inv cache.

- **Read hit**: Never entails mode changes.

- **Write miss to an inv cache**: That cache will become mdf. At the same time, all other caches storing A will become inv

$$(S_{\mathtt{A}}, M_{\mathtt{A}}) \to (0, 1)$$

- **Write miss to a shr cache storing A**: As this is a "miss" case, the cache will cease to count on $S_{\mathtt{A}}$. We get the same transition as in the corresponding case for a read.

- **Write miss to a shr cache that will begin to store A**: Its copy will become the only valid one, according to the same transition we got for a write miss to an inv cache.

- **Write miss to a mdf cache storing A**: That cache will cease to count on $M_{\mathtt{A}}$. The resulting transition is not new, but the same we got for a read miss to a mdf cache storing A.
- **Write miss to a mdf cache that will begin to store A**: Its copy will become the only valid one, as it was in the case of a write miss to an inv cache.
- **Write hit**: Being a hit, there will not be an address change in the cache. But, after writing, that will be the only cache with a valid copy, as in the previous item.

In brief, and translating to Maude:

```
rl [a] : (S, 0) => (s(S), 0) .
rl [b] : (S, s(M)) => (s(s(S)), M) .
rl [c] : (s(S), M) => (S, M) .
rl [d] : (S, s(M)) => (S, M) .
rl [e] : (S, M) => (0, 1) .
```

Now, we define:

```
op coherent : -> StateProp [ctor] .
eq (S, M) |= coherent = (M == 0 or (M == 1 and S == 0)) .
```

And issue this command:

```
(ismc [50] (0, 0) |= exists F NOT coherent .)
```

As expected, it does not find a witness.

This is a quite simple system, but not trivial: visual inspection is probably not enough to convince oneself that $M_{\mathtt{A}} > 0$ implies $S_{\mathtt{A}} = 0$. It is surely possible to define some abstraction that turns the system finite. But the result of the model checker, if not conclusive, might be reassuring enough, even if only as a first, preliminary test. As discussed in Section 4.1, searching from the very beginning an appropriate abstraction and proving it fitting may not be worth the effort, because this simple analysis could have found a flaw, or can be convincing enough.

Chapter 6

Related Work

We comment some work related to four subjects dealt with in this master thesis, and finish with some proposals for future work.

6.1 Model Checking TLR*

The papers [6, 3, 4, 5] are all related to model checking Maude modules with LTLR formulas. The logic LTLR is the linear sublogic of TLR*, that is, formulas with no path quantifiers taken to be universally quantified on paths.

In [6] the authors implement a translation, already described in [21], that allows the use of Maude's LTL model checker. The idea is the following: We are given a rewriting system $\mathcal{R}$, with an initial state on it, and an LTLR formula as parameters to perform model checking on them. From $\mathcal{R}$ we produce a new system, equivalent to $\mathcal{R}$ in an appropriate way, whose states store, in addition to its own information, also data on the transition that took the system to them. In parallel, we translate the given LTLR formula to an LTL formula with equivalent semantics. The result shown on [21] is that this simultaneous mapping of systems and formulas can be done in a *faithful* way, that is, the produced system models the produced formula iff the given system models the given formula. Thus, as the final step of the implementation, Maude's LTL model checker is internally used. All this is done within Maude, extending Full Maude, as we have done for our own tool.

Looking for better performance, [3] implements the LTLR model checker in C++, modifying the implementation of Maude's LTL model checker.

In model checking, fairness constraints are often needed as a natural precondition for some temporal properties to hold. Typical fairness constraints are not expressible in linear temporal logic. The papers [4, 5] show how fairness constraints can be included in the system specification, and how model checking LTL or LTLR formulas can be done taking these constraints into account in the very algorithm. Moreover, these papers show how to use *parameterized* fairness properties, that allow the user to specify which entities of the system have to be treated with fairness and which others we do not care about.

6.2 Infinite Systems

Model checking on infinite systems has been the subject of a lot of papers. Most of them look for ways to either find an abstraction that turns the system finite, or find a way to finitely represent the elements that compose the system. To the best of our knowledge, the only previous try to do it explicit-state is in [25, 26]. Both of them use Maude for system specification and μ-calculus to express temporal properties of the system.

There are several possible sources of infinity in a system. For instance, some object in the system, like a counter, might be unbounded. Or the amount of objects in the system might be increasing, also in an unbounded way. Each kind of infinity may be amenable to different techniques. We very briefly mention next a few of the approaches we know of.

Abstraction is a well-known mechanism to make the size of a system smaller, where *smaller* can even mean finite from infinite. See [22] for an approach within rewriting logic. The idea of abstraction is to group together states that, though different, are indistinguishable to the formula we try to model check. In more technical terms, the aim is to find a finite system that is bisimilar to the given infinite one. If the formula, for instance, only depends on a counter being zero or not, one can collapse all nonzero values of the counter to just one value. In this respect, model checking on timed systems often uses *time regions* with the same idea: instead of using time *instants*, use well chosen time *intervals*, taking care that the given temporal formula is not able to tell apart two instants on the same interval.

In the way of finite representability, the method of *well-structured transition systems* has proven useful [1, 16]. A well-structured transition system is one in whose infinite set of states a well-quasi ordering has been defined. A well-quasi ordering is a quasi-ordering (that is, a reflexive and transitive relation) such that no infinite strictly decreasing sequence exists. In a well-structured system certain sets (so called *upward-closed sets*) of states are finitely representable. These sets are enough to provide algorithms to solve many model-checking problems. The reference [16] lists a collection of natural examples for which a well-quasi ordering can be found. Among others, it includes string rewriting systems of the kind we briefly studied in Section 4.3, and defines four different well-quasi orderings on them.

6.3 Model Checking Cache Coherence Protocols

As hardware architectures get more and more complex, tools for the automatic verification of cache coherence protocols become more and more convenient. We review two of the published proposals. Both deal with *parameterized* verification, that is, they intend to verify a protocol for any number of caches. The means to achieve that is to apply a *counting abstraction*, very similar to what we did with our level 3 model in Section 5.5.

In [12] Delzanno uses linear arithmetic constraints to finitely represent infinite sets of states, where *states* must be understood as tuples resulting from the counting abstraction. The dynamics of the system, that is, the transitions between tuples, are formalized using "extended finite states machines"—finite-state machines that include the use of variables within states and global conditions for transitions. Among other examples, it shows how to

model check the Synapse N+1 protocol, very similar to MSI, focusing on safety properties.

Also [14] uses MSI as "motivating example." Its abstraction does not rely just on counting. Instead of working with transitions between cache modes mdf, shr, and inv, as we have sketched in Section 5.2, it uses transitions between a kind of extended system states, that take into account the possible modes of both the latest cache acted upon and the rest of caches. For instance, (inv, {inv,shr}) means that *one* cache is in mode inv while all others are in modes inv or shr. From this extended state, there is a transition to (mdf, {inv}), because when *this* cache changes from inv to mdf all others in mode shr must be invalidated.

6.4 Strategies

Strategies do not seem to have been used as a means to model checking before. However, they are present in several languages. In Maude, there is a rich strategy language; see [11, 19], for instance. In some sense, that is a more powerful strategy language than the one we present here, although none of them contains the other. In particular, Maude's strategy language does not include an *until* operator, but it does include constructs to control the rewriting of subterms within the main term being rewritten.

In [24] strategies are used in the framework of program transformation (like for refactoring, compiling, optimization). In particular, the Stratego language is used. Stratego is a language for program transformation based on rewriting and strategies. ELAN, described for instance in [7], is a rewriting logic language. Both ELAN and Stratego have strategies included in the language, while in Maude system modules and strategy modules are separated syntactic entities.

Chapter 7

Conclusions and Future Work

Several subjects related to system specification and verification have got roles in this work: rewriting logic (and Maude) as a specification formalism, rewriting logic (and Maude) as a software development tool, state-based and action-based temporal logics and TLR*, guarantee and safety properties, strategies applied to nondeterministic systems, and model checking on infinite systems. We have introduced all of them. We have implemented a strategy language and shown how it can be used to model check TLR* guarantee formulas on possibly infinite systems by first translating them into strategy expressions. Finally, we have proven the usefulness of the tool on a series of examples, with particular emphasis on verifying the MSI cache coherence protocol.

An explicit-state model-checking procedure on infinite systems cannot be expected to produce a definitive answer in all cases, and cannot be expected either to provide the best performance. However, the point is that our model-checking procedure is available almost for free as soon as one has a system specified. Quoting again Meseguer, talking about the example presented in Section 4.1, "all such efforts to obtain a tractable finite-state abstraction, and the associated theorem proving work to check confluence, coherence and preservation of state predicates for the abstraction, are not even worth it; since this simpler analysis of the system specification has already uncovered a key flaw." Thus, we think explicit-state model checking deserves a place in an infinite-system verification toolbox.

Several improvements and lines for additional work are possible. A C++ implementation in search for better performance is one of the most obvious things to do.

In a different line, the whole strategy language can be made available to the user. As things are now, strategies are only used internally, and the user can only write TLR* guarantee formulas, that are a proper subset of the strategy expressions. However, the whole strategy language is already implemented, and only the routine, user-interface part remains to be coded.

Maybe more interestingly, some additional improvements can be added to the model checker. For instance, we already have loop detection, that is, detection of repeated (state, strategy) pairs on the same path. But, when repetition occurs in different branches, we

are not ready to detect it. For some systems, this would provide a drastically improved performance; but, for systems with no repetitions, the impact would be negative. Thus, a `set` command to enable or disable this feature would be necessary.

More or less in this same line, abstraction tools can be offered to the user. That is, a means can be implemented to allow the user specify when two different states can be considered equivalent to the model-checking task being currently performed. That would be used to improve both loop detection and different-branch repetition detection. It would be an optional feature, of course.

Going a step further, the concept of well-structured transition system, referred to in Section 6.2, can be taken into account. To this goal, the user would have to specify the quasi-ordering relation between states. This line of work seems specially interesting, as well-structured systems have been an active field for more than a decade now and, to the best of our knowledge, no work has related them to rewriting logic.

Finally, and also looking for better performance and usefulness, a tool for *partial order reduction* can be implemented to allow the user specify when two transitions are independent, so that only one way to order and perform them must be taken into account. This way, whole paths in the system's state space are avoided from their roots. The reference [15], for instance, has proposals on how to implement partial order reduction.

Bibliography

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE Computer Society, 1996.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Kyungmin Bae and José Meseguer. The linear temporal logic of rewriting Maude model checker. In Peter Csaba Ölveczky, editor, *Rewriting Logic and its Applications. 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2010.

[4] Kyungmin Bae and José Meseguer. State/event-based LTL model checking under parametric generalized fairness. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 132–148. Springer, 2011.

[5] Kyungmin Bae and José Meseguer. Model checking LTLR formulas under localized fairness. In Francisco Durán, editor, *Rewriting Logic and Its Applications - 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers*, volume 7571 of *Lecture Notes in Computer Science*, pages 99–117. Springer, 2012.

[6] Kyungmin Bae and José Meseguer. A rewriting-based model checker for the linear temporal logic of rewriting. *Electr. Notes Theor. Comput. Sci.*, 290:19–36, 2012.

[7] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[8] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[9] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical*

Framework, How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[11] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 126–148. Elsevier, 1996.

[12] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[13] Francisco Durán and José Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, 2000.

[14] E. Allen Emerson and Vineet Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 144–159, Berlin, Heidelberg, 2003. Springer-Verlag.

[15] Azadeh Farzan. *Static and Dynamic Formal Analysis of Concurrent Systems and Languages: A Semantics-Based Approach*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[16] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[17] Jim Handy. *The Cache Memory Book*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Academic Press Inc., 1998.

[18] Narciso Martí-Oliet, editor. *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[19] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. In Martí-Oliet [18], pages 417–441.

[20] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[21] José Meseguer. The temporal logic of rewriting. Technical Report UIUCDCS-R-2007-2815, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[22] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.

[23] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126, March 1997.

[24] Eelco Visser. A survey of strategies in program transformation systems. *Electr. Notes Theor. Comput. Sci.*, 57:109–143, 2001.

[25] Bow-Yaw Wang. μ-calculus model checking in Maude. In Martí-Oliet [18], pages 135–152.

[26] Bow-Yaw Wang. Specification of an infinite-state local model checker in rewriting logic. In William C. Chu, Natalia Juristo Juzgado, and W. Eric Wong, editors, *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, Taipei, Taiwan, Republic of China, July 14-16, 2005*, pages 442–447, 2005.

[27] Wikipedia. Formal grammar — wikipedia, the free encyclopedia, 2013. [Online; accessed 29-May-2013].