Towards Generic Monitors for Object-Oriented Real-Time Maude Specifications

Antonio Moreno-Delgado[1], Francisco Durán[1], José Meseguer[2]

[1]University of Málaga, Spain.
[2]University of Illinois at Urbana-Champaign, United States.
{amoreno,duran}@lcc.uma.es, meseguer@illinois.edu

Abstract. Non-Functional Properties (NFPs) are crucial in the design of software. Specification of systems is used in the very first phases of the software development process for the stakeholders to make decisions on which architecture or platform to use. These specifications may be analyzed using different formalisms and techniques, simulation being one of them. During a simulation, the relevant data involved in the analysis of the NFPs of interest can be measured using monitors. In this work, we show how monitors can be parametrically specified so that the instrumentation of specifications to be monitored can be automatically performed. We prove that the original specification and the automatically obtained specification with monitors are bisimilar by construction. This means that the changes made on the original system by adding monitors do not affect its behavior. This approach allows us to have a library of possible monitors that can be safely added to analyze different properties, possibly on different objects of our systems, at will.

1 Introduction

As system complexity grows, specification of systems becomes an even more important task during the first phases of the software life cycle. With the proliferation of distributed systems due to Cloud-computing systems, Internet of Things, etc., with software being present in all activities of our lives, Non-Functional Properties (NFPs) are gaining relevance in design decisions.

Specification of software and its simulation can be used to get insights about how the system is going to behave. Furthermore, by adding monitors or observers to system specifications, software engineers can analyze those NFPs of interest [9]. System specifications have to be instrumented in order to get probes of executions. One may think of different NFPs, such as response time, throughput, mean cycle time or rate of failures. However, different NFPs have to be monitored by different observers, and such observers are typically hard-coded in the specifications.

To cope with this lack of modularity, many alternatives have been proposed. For example, in Aspect-Oriented Programming, code is instrumented by monitors as a cross-cutting concern. Other works, as the one presented in [16], propose adding observers as new elements (objects) of the language. See [17] for a discussion of how to monitor non-functional properties in component-based systems.

In most cases, and even with more emphasis in the case of distributed or concurrent systems, these specifications are written with, among others, the purpose of verification. Different kinds of verification can be achieved depending on the desired level of evidence and precision in the proofs. Furthermore, some formalisms are more amenable to perform some proofs or checks than others. For example, a specification in Promela/Spin [8] is more amenable to perform model-checking that a specification in UML. Likewise, a specification in Coq [1] is well-suited to perform theorem-proving. This means that a tight dependence between formalisms and the verification one can perform exists. Among all kind of formalisms, we find very attractive those which can be executed, since the software engineers involved in the software development can get insights on where they are failing or on which parts they have to stress.

Execution of a system specification means that the system at hand can be simulated in the very first phases of software design, and, at very low additional cost, software engineers can test different designs and approaches, thus getting insight about how the system is going to fulfill the required NFPs. However, to analyze the behavior of a system under simulation, we have to measure the properties we want to study.

Rewriting logic [10] provides a formal framework where concurrent and distributed systems can be naturally defined. Since the specification remains within a formal environment, different kinds of verification can be performed: confluence, model-checking, reachability analysis or invariant analysis. Additionally, rewriting logic specifications are executable, providing prototypes that can be simulated and tested.

In this work we propose the definition of monitors in a very general way. If monitors are defined following certain guidelines, their addition to any real-time object-oriented specification is automatic, and what more important, the original behavior of the system after being instrumented is not changed.

We focus on object-oriented modules that must be defined using Real-Time Maude [14], since the main applications we envision are real-time and stochastic systems. On these specifications, we are interested in measuring system properties, i.e., properties that affect the whole system as throughput, and individual properties, i.e., properties related to concrete objects as traffic or utilization.

Monitors can be defined just by querying data. Thus, we give a skeleton Maude module which can be used to define any kind of monitor query by specifying the data structure to use and the query to perform.

Besides the theoretical results, a tool in the rewriting logic language Maude is presented to include generically defined monitors to system specifications. Using the reflective capabilities of Maude, we have defined module operations that take the specifications to be analized and the generic monitors to be used on them, and generate new modules with the instrumented specifications. We have used the extensibility capabilities of Full Maude [4, 6] to provide a new module expression giving access to such module operation. Thus, we can not only automatically instrument our specifications with reusable monitors, but also use them in our specifications and commands as any other module.

The rest of the paper is structured as follows. Section 2 presents the rewriting logic language Maude and its Real-Time Maude extension, which allows us to define systems with time annotations. Section 3 presents the structure of monitors we use and basic principles of the approach. Section 4 presents the automatic transformation and the module operation implementing it. Section 5 provides the proof for bisimilarity between the original specification and the instrumented one. Section 6 provides one additional example. Section 7 wraps up the paper with some conclusions and ideas for further extensions and improvements.

2 Maude and Real-Time Maude

Maude [2, 3] is an executable formal specification language based on rewriting logic [10], a logic of change that can naturally deal with states and non-deterministic concurrent computations. A rewrite logic theory is a tuple (Σ, E, R), where (Σ, E) is an *equational theory* that specifies the system states as elements of the initial algebra $\mathcal{T}_{(\Sigma,E)}$, and R is a set of rewrite rules that describe the one-step possible concurrent transitions in the system.

Rewriting operates on congruence classes of terms modulo E. This of course does not mean that an implementation of rewriting logic must have an E-matching algorithm for each equational theory E that a user might specify. The equations E are divided into a set A of structural axioms for which matching algorithms are available and a set E of equations. Then, for having a complete agreement between the specification's initial algebra and its operational semantics by rewriting, a rewrite theory $(\Sigma, E \cup A, R)$ is assumed to be such that the set E of equations is (ground) Church-Rosser and terminating modulo A, and the rules R are (ground) coherent with the equations E modulo A (see [5, 7]).

Maude provides support for rewriting modulo associativity, commutativity and identity, which perfectly captures the evolution of systems made up of objects linked by references. Maude has a rich set of verification and validation tools, and its use is widespread in many fields of research. Furthermore, Maude has demonstrated to be a good environment for rapid prototyping, and also for application development (see [3]).

Among the tools and extensions of Maude, one interesting tool for specifying distributed and concurrent systems is Real-Time Maude [14], a rewriting-logic-based specification language and formal analysis tool that supports the formal specification and analysis of *real-time systems*. Real-Time Maude provides a sort `Time` to model the time domain, which can be either discrete or dense. Then, passage of time is modelled with *tick rules* of the form

$$\texttt{crl [}l\texttt{] : \{ } t\texttt{, } T \texttt{ \} => \{ } t'\texttt{, } T+\tau \texttt{ \} if } C \texttt{ .}$$

where t and t' are system states, T is the global time, and τ is a term of sort `Time` that denotes the *duration* of the rewrite, and that advances by τ the *global time elapse*. Since tick rules advance the global time, in Real-Time Maude time elapse is usually modeled by one single tick rule, and the system dynamic behavior by instantaneous transitions [14]. Although there are other sampling strategies, in the most convenient one this single tick rule models time elapse by using two functions: the `delta` function, that defines the effect of time elapse over every model element, and the `mte` (maximal time elapse) function, that defines the maximum amount of time that can elapse before any action can be performed. Then, time can advance non-deterministically by any time amount τ, which must be less than or equal to the maximum time elapse of the system.

crl [tick] : { t, T } => { delta(t, τ), $T+\tau$ } if $0 < \tau \leq$ mte(t) $\wedge$ C .

3 General monitors

In this section we present our proposal for the specification of system-independent monitors. Given a Real-Time Maude object-oriented system specification we provide operations to automatically add objects to measure different properties. We distinguish two types of properties, namely, those on individual objects, e.g., the number of messages received by each node in a network, or the number of defective pieces produced by each machine in a production line, and those on global systems, e.g., the average time taken by messages in reaching their destination or the average failure rate of the machines in a system. We handle both cases uniformly by assuming that there are classes in our specification whose objects "represent" the subsystems being monitored. For instance, we might assume that our network of nodes has a *net* object with references to all the nodes in it. This would allow us to use an individual monitor associated to the net object instead of a system monitor associated to all the node objects. This might be the case if we wanted to consider, for instance, multiple nets in the same system and separately monitor information on them.

We assume a Real-Time Maude object-oriented specification, with a flat configuration of objects and messages (i.e., no nested configurations) and with all rewrite rules of the system defined on terms of sort `System`, that is, on terms of the form { *Conf*, *T* }, with *Conf* a flat configuration and *T* a term of sort `Time`.

To present and illustrate our monitors, we use a very simple specification of a messaging system, shown in Figure 1, where we have interconnected nodes, some of which belong to a subclass a message creator nodes, which create messages to be delivered through the net via specific neighbors. The `Node` class is defined with an attribute `neighbors` of type `List{Oid}`. Its `MsgCreator` subclass has, in addition, attributes `targets`, with the identifiers of the nodes it may be addressing messages to, and a `counter` to limit the number of generated messages. The `Net` class represents the entire net of nodes. It has an attribute `elems` with the identifiers of the nodes in the net. Messages are of the form `to T via N`, without sender identifier nor any contents to simplify the specification, where `T` is the identifier of the target node and `N` is the neighbor node the message is being sent through. The auxiliary operation `pickOne` is used to select an element in a list,

```
omod SMP is
 pr NAT-TIME-DOMAIN-WITH-INF .
 inc RANDOM + COUNTER .
 pr LIST{Oid} .

 var  Msg : Msg .                    vars O O1 ON : Oid .
 var  VCreator : MsgCreator .        var  VNode : Node .
 var  VNet : Net .                   vars T T' : TimeInf .
 vars L L' EL EL' : List{Oid} .      var  N : Nat .
 var  Atts : AttributeSet .          var  Conf : Configuration .

 sort System .
 op {_,_} : Configuration TimeInf -> System [ctor] .

 class Net | elems : List{Oid} .
 class Node | neighbors : List{Oid} .
 class MsgCreator | targets : List{Oid}, counter : Nat .
 subclass MsgCreator < Node .
 msg to_via_ : Oid Oid -> Msg .

 op delay : Msg Time -> Msg .
 eq delay(Msg, 0) = Msg .

 rl [create-msg] :
   { < ON : VNet | elems : (EL O EL') >
     < O : VCreator | targets : L, neighbors : L',
        counter : s(N), Atts > Conf, T }
   =>
   { < ON : VNet | elems : (EL O EL') >
     < O : VCreator | targets : L, neighbors : L', counter : N, Atts >
     delay(to pickOne(L, random(counter) rem size(L))
              via pickOne(L', random(counter) rem size(L')),
           random(counter) rem 500)
     Conf, T } .
 rl [get-msg] : { < ON : VNet | elems : (EL O EL') >
                  < O : VNode | Atts > (to O via O1) Conf, T }
   => { < ON : VNet | elems : (EL O EL') >
        < O : VNode | Atts > Conf, T } .
 crl [resend-msg] : { < ON : VNet | elems : (EL O EL') >
                      < O : VNode | neighbors : L, Atts >
                      (to O1 via O) Conf, T }
   => { < ON : VNet | elems : (EL O EL') >
        < O : VNode | neighbors : L, Atts >
        delay(to O1 via pickOne(L, random(counter) rem size(L)),
              random(counter) rem 5) Conf, T }
   if O =/= O1 .

 op pickOne : List{Oid} Nat ~> Oid .
 eq pickOne(O L, 0) = O .
 eq pickOne(O L, s(N)) = pickOne(L, N) .

 op mte : Configuration -> TimeInf .
 eq mte(delay(Msg, T) Conf) = min(T, mte(Conf)) .
 eq mte((to O via O1) Conf) = 0 .
 eq mte(Conf) = INF [owise] .

 op delta : Configuration Time -> Configuration .
 eq delta(delay(Msg, T) Conf, T')
   = delay(Msg, T monus T') delta(Conf, T') .
 eq delta(Conf, T) = Conf [owise] .

 crl [tick] : { Conf, T } => { delta(Conf, T'), T + T' }
   if T' := mte(Conf) /\ 0 < T' /\ T' < INF .
endom
```

Fig. 1. Specification of a simple messaging system

```
omod MONITOR is
 pr CONFIGURATION .
 pr NAT-TIME-DOMAIN-WITH-INF .
 sort Data .
 class @Monitor | o : Object , data : Data .
 op eval : Data Time Object Configuration Configuration Configuration
      -> Data .
 op mon : Oid -> Oid [ctor] .
endom
```

Fig. 2. Core of monitors

which will be used in the `create-msg` and `resend-msg` rules to randomly select elements in the list of targets and neighbors. The `create-msg` rule creates a new message addressed to a random target via a random neighbor, the `get-msg` specifies the reception of a message by its addressee, and the `resend-msg` rule specifies the action in which a node receives a message that is not addressed for it and resends it via one of its neighbors. Note that such rule will resend the message via one of its randomly chosen neighbors. Delays in message delivery is specified with the usual `delay` operator (see [14]). Real-Time Maude's `tick` rule and `mte` and `delta` functions are defined as usual.

Inspired by the works on wrapper objects, and specifically on the Onion-Skin pattern [13, 11], we add monitors to our specification by means of wrappers. We will show a generic monitor structure that, by specifying the definition of the *data structure* and the query for the monitor to use, can be instantiated to a concrete monitor to be added to our system.

Each object to be monitored is wrapped inside a monitor object that will observe its behavior and will collect the required information on it. This generic monitor structure is defined by the `MONITOR` module in Figure 2. There is a class `@Monitor` whose instances will wrap objects in their `o` attributes. The data of the monitor is stored in the attribute `data`, of sort `Data`, to be later instantiated depending on the specific kind of monitor defined. There is an operation `eval`, that will be used to recalculate the monitored information, depending on the actions specified in individual rules, with parameters: (i) the current monitor's data, (ii) the time at which the expression is evaluated, (iii) the monitored object in the LHS of the rule, (iv) the objects and messages explicitly stated in the rule's LHS, (v) the objects and messages in the rule's RHS, and (vi) the rest of the LHS' configuration. The individual monitor of an object with identifier O will have identifier `mon(`O`)`.

Specific monitors can be defined by specifying of the function *eval*, which could be defined over any data structure, just by appropriately subsorting the sort *Data*. For example, given the simple messaging system specified in the module in Figure 1, we may count the number of messages received by each of the nodes in the system by wrapping each of them inside monitor objects as in Figure 4, and by defining the `eval` function in a module `TRAFFIC-MONITOR` extending the `MONITOR` module, given an auxiliary `#msgs` function which counts the number of messages in a configuration, as shown in Figure 3. Note that the `data` attribute remains unchanged in the `create-msg` rule, but it is recalculated in rules `get-msg` and `resend-msg`, those rules in which node objects receive messages.

```
omod TRAFFIC-MONITOR is
 inc MONITOR .
 pr NAT .
 subsort Nat < Data .

 var  N : Nat .
 var  T : Time .
 var  Obj : Object .
 vars LConf RConf GConf : Configuration .

 eq eval(N, T, Obj, LConf, RConf, GConf)
   = N + #msgs(LConf) .

 op #msgs : Configuration -> Nat .
 eq #msgs(Msg Conf) = s(#msgs(Conf)) .
 eq #msgs(Conf) = 0 [owise] .
endom
```

Fig. 3. Traffic monitors

The `subsort` relation states the data type of the monitor data. This monitor is going to store only a natural number, used to count the number of messages the node at hand has processed. Note that the operation `eval` is total and it will increment the natural number stored in the monitor with the number of messages in the rule's LHS.

By rewriting our initial configuration with our nodes wrapped inside monitor objects using the rules in Figure 4, we get a final configuration in which the `data` attributes of each of the monitor objects contains the number of messages received by that node.

4 Construction of the instrumented specification

The construction of instrumented specifications has been automated by providing a module expression `MONITOR` that takes as arguments the specification to be monitored, the class whose objects are to be wrapped, the set of rules on which the measures are to be evaluated, and a concrete monitor to apply to it, in which the `Data` sort and the `eval` functions are defined, and that produces the corresponding new module. The module expression is integrated in Full Maude and is handled as any other module expression [6].

Given an object-oriented system specification S, a class C, a set of rule labels LS, and a concrete monitor M, the rewrite theory $M[S, C, LS, E]$ denotes the system S but now instrumented with the monitor E as follows:

- $M[S, C, LS, E]$ includes both S and M, plus transformed copies of the rules of S so that each rule of the form

```
var  Msg : Msg .                      vars O O1 ON : Oid .
var  VCreator : MsgCreator .          var  VNode : Node .
vars L L' EL EL' : List{Oid} .        var  N : Nat .
vars Atts @Atts : AttributeSet .      var  Conf : Configuration .
vars T T' : TimeInf .                 var  VNet : Net .
var  @D : Data .

rl [create-msg] :
  { < ON : VNet | elems : (EL O EL') >
    < mon(O) : @Monitor |
       o : < O : VCreator |
                targets : L, neighbors : L', counter : s(N) > >
    Conf , T }
  =>
  { < ON : VNet | elems : (EL O EL') >
    < mon(O) : @Monitor |
       o : < O : VCreator |
                targets : L, neighbors : L', counter : N > >
    delay(to pickOne(L, random(counter) rem size(L))
             via pickOne(L', random(counter) rem size(L')),
          random(counter) rem 500)
    Conf , T } .
rl [get-msg] : { < ON : VNet | elems : (EL O EL') >
                 < mon(O) : @Monitor | o : < O : VNode | Atts > >
                 (to O via O1) Conf, T }
  => { < ON : VNet | elems : (EL O EL') >
       < mon(O) : @Monitor | o : < O : VNode | Atts > > Conf, T } .
crl [resend-msg] : { < ON : VNet | elems : (EL O EL') >
                     < mon(O) : @Monitor |
                        o : < O : VNode | neighbors : L >,
                        data : @D >
                     (to O1 via O) Conf, T }
  => { < ON : VNet | elems : (EL O EL') >
       < mon(O) : @Monitor |
          o : < O : VNode | neighbors : L >,
          data : eval(@D, T,
                   < O : VNode | neighbors : L >,
                   (< ON : VNet | elems : (EL O EL') >
                    < O : VNode | neighbors : L > (to O1 via O)),
                   (< ON : VNet | elems : (EL O EL') >
                    < O : VNode | neighbors : L >
                    delay(to O1
                          via pickOne(L, random(counter) rem size(L)),
                          random(counter) rem 5)),
                   Conf) >
       delay(to O1 via pickOne(L, random(counter) rem size(L)),
             random(counter) rem 5) Conf, T }
  if O =/= O1 .
```

Fig. 4. Rules of the simple messaging system with individual monitors

```
crl [L] : { < O : C' | Atts > Conf , T }
  => { < O : C' | Atts' > Conf' , T }
   if Cond .
```

with C' a subclass of C or C itself, and L in LS, generates a new rule

```
crl [L] :
  { < mon(O) : Monitor | o : < O : C | Atts >, data : D > Conf , T }
  =>
  { < mon(O) : Monitor |
        o : < O : C | Atts' >,
        data : eval(D, T,
                    < O : C | Atts >,
                    Conf ,
                    < O : C | Atts' > Conf') >
    Conf' ,
    T }
   if Cond .
```

– All other occurrences of objects

```
< O : C | Atts >
```

of subclasses of C in rules, equations and memberships will be rewritten as

```
< mon(O) : Monitor | o : < O : C | Atts >, data : D >.
```

– All other objects in rules are left as they were.
– In case multiple objects appear in the same rule/equation/membership, different D variables will be consistently used. E.g., if L is not in LS, for a rule with two objects of class C in its left-hand side, the following rule will be generated:

```
crl [L] :
  { < mon(O1) : Monitor | o : < O1 : C | Atts1 >, data : D1 >
    < mon(O2) : Monitor | o : < O2 : C | Atts2 >, data : D2 >
    Conf , T }
  =>
  { < mon(O1) : Monitor | o : < O1 : C | Atts1' >, data : D1 >
    < mon(O2) : Monitor | o : < O2 : C | Atts2' >, data : D2 >
    Conf' , T }
```

Note that:

– Those rules with no objects in subclasses of C remain as in the original module, and
– There might be more than one object in subclasses of C in the lefthand side of a rule, in which case the above transformation has to be applied to each of them, that is, we must consider all possible matches of the above pattern. E.g., given a rule

```
crl [L] :
  { < O1 : C1 | Atts1 >
    < O2 : C2 | Atts2 >
    Conf , T }
  =>
  { < O1 : C1 | Atts1' >
    < O2 : C2 | Atts2' >
    Conf' , T }
   if Cond .
```

with $C1$ and $C2$ subclasses of C and L in LS, we get the rule

```
crl [L] :
  { < mon(O1) : Monitor | o : < O1 : C1 | Atts1 >, data : D1 >
    < mon(O2) : Monitor | o : < O2 : C2 | Atts2 >, data : D2 >
    Conf , T }
 =>
  { < mon(O1) : Monitor |
        o : < O1 : C1 | Atts1' >,
        data : eval(D1, T,
                    < O1 : C1 | Atts1 >,
                    < O2 : C2 | Atts2 > Conf ,
                    < O1 : C1 | Atts1' > < O2 : C2 | Atts2' > Conf') >
    < mon(O2) : Monitor |
        o : < O2 : C2 | Atts2' >,
        data : eval(D2, T,
                    < O2 : C2 | Atts2 >,
                    < O1 : C1 | Atts1 > Conf ,
                    < O1 : C1 | Atts1' > < O2 : C2 | Atts2' > Conf') >
    Conf' , T }
  if Cond .
```

Given the `SMP` module shown in Figure 1 and the module `TRAFFIC-MONITOR` in Figure 3 defining the counter of received messages, the module expression

```
MONITOR[SMP, Node, get-msg resend-msg, TRAFFIC]
```

produces the instrumented version of the `SMP` module as previously explained. This module operation is indeed integrated in Full Maude and can be used, for example, to execute the following rewrite command:

```
rew in MONITOR[SMP, Node, get-msg resend-msg, TRAFFIC-MONITOR] :
 { < n  : Net | elems : (n1 n2 n3 n4 n5 n6 n7) >
   < mon(n1) : @Monitor |
      o : < n1 : MsgCreator | targets : (n2 n3 n4 n5 n6 n7),
                              neighbors : (n2 n3 n4 n5 n6),
                              counter : 500 >,
      data : 0 >
   < mon(n2) : @Monitor | o : < n2 : Node | neighbors : (n1 n3 n7) >,
                          data : 0 >
   < mon(n3) : @Monitor | o : < n3 : Node | neighbors : (n1 n2 n4) >,
                          data : 0 >
   < mon(n4) : @Monitor | o : < n4 : Node | neighbors : (n1 n3 n5) >,
                          data : 0 >
   < mon(n5) : @Monitor | o : < n5 : Node | neighbors : (n1 n4 n6) >,
                          data : 0 >
   < mon(n6) : @Monitor | o : < n6 : Node | neighbors : (n1 n5 n7) >,
                          data : 0 >
   < mon(n7) : @Monitor | o : < n7 : Node | neighbors : (n1 n6 n2) >,
                          data : 0 >, 0 } .
result GoodSystem :
 { < n : Net | elems : (n1 n2 n3 n4 n5 n6 n7) >
   < mon(n1) : @Monitor |
      o : < n1 : MsgCreator | neighbors : (n2 n3 n4 n5 n6),
                              targets : (n2 n3 n4 n5 n6 n7),
                              counter : 0 >,
    data : 923 >
   < mon(n2) : @Monitor | o : < n2 : Node | neighbors : (n1 n3 n7) >,
                          data : 459 >
   < mon(n3) : @Monitor | o : < n3 : Node | neighbors : (n1 n2 n4) >,
                          data : 545 >
   < mon(n4) : @Monitor | o : < n4 : Node | neighbors : (n1 n3 n5) >,
                          data : 537 >
   < mon(n5) : @Monitor | o : < n5 : Node | neighbors : (n1 n4 n6) >,
                          data : 530 >
```

```
  < mon(n6) : @Monitor | o : < n6 : Node | neighbors : (n1 n5 n7) >,
                         data : 470 >
  < mon(n7) : @Monitor | o : < n7 : Node | neighbors : (n1 n6 n2) >,
                         data : 219 >,
  1238 }
```

5 Addition of individual monitors preserves behavior

Adding individual monitors to our specification should not modify the behavior of the system specification, in the sense that there must be a one-to-one correspondence between the rewrites in the original specification and the instrumented one. This idea is captured by the notion of bisimulation, defined as a simulation relation whose inverse relation is also a simulation [12]. In this section we provide bisimulation proofs for the addition of monitors.

We will name S a generic system defined as an object-oriented system with time annotations. We assume a Real-Time Maude specification as above described. We will denote by E a particular monitor to be added to S. The result of the composition of E in S, with a distinguish class C of S and a set of labels of rules LS of S, will be denoted as $M[S, C, LS, E]$. In this section we prove that adding this general individual monitors does not modify the behavior of our system by showing that a bisimulation between $M[S, C, LS, E]$ and S exists.

First, notice that the transformation injecting the monitors depends on the class whose objects are to be monitored, and that in order to define a total function we need to restrict the kind of systems we may consider. We introduce sorts $\mathit{GoodSystem}_S$ and $\mathit{GoodSystem}_{M[S,C,LS,E]}$ respectively as subsorts of System_S and $\mathit{System}_{M[S,C,LS,E]}$. The kind of object configurations permitted in these sorts satisfy all the usual requirements of object configurations (no repeated object identifiers, no repeated attributes in objects, objects have attributes defined in their classes or superclasses, etc.). Moreover, all objects in configurations of terms of sort $\mathit{GoodSystem}_{M[S,C,LS,E]}$ wrapped in monitor objects are instances of class C or subclasses of it. We define these good-system sorts using conditional memberships.

By using techniques related to ground invariance [15], and assuming that the term algebra $\mathcal{T}_{\Sigma_S/E_S,\mathit{GoodSystem}_S}$ is closed under the relation $\rightarrow_{\mathcal{R}_S}$, we prove that $\mathcal{T}_{\Sigma_{M[S,C,LS,E]}/E_{M[S,C,LS,E]},\mathit{GoodSystem}_{M[S,C,LS,E]}}$ is closed under $\rightarrow_{\mathcal{R}_{M[S,C,LS,E]}}$. If not total, a transition relation $\rightarrow$ can be extended to $\rightarrow^\bullet$ by adding pairs of the form $a \rightarrow^\bullet a$ when a cannot be rewritten (see [12, 3] for an automatic transformation). Assuming a set of propositions AP and labeling functions $L_S : \mathit{GoodConfig}_S \rightarrow P(AP)$ and $L_{M[S,C,LS,E]} : \mathit{GoodConfig}_{M[S,C,LS,E]} \rightarrow P(AP)$, which associates to each state with the set of atomic propositions that hold in it, we extend S and $M[S, C, LS, E]$ to Kripke structures $\mathsf{A}_S = (\mathcal{T}_{\Sigma_S/E_S,\mathit{GoodSystem}_S}, \rightarrow_S, L_S)$ and $\mathsf{A}_{M[S,C,LS,E]} = (\mathcal{T}_{\Sigma_{M[S,C,LS,E]}/E_{M[S,C,LS,E]},\mathit{GoodSystem}_{M[S,C,LS,E]}}, \rightarrow_{M[S,C,LS,E]}, L_S \circ H)$, respectively, where H is the function defined below.

Let us consider the following map H and let us prove it is a (strict) simulation:

$$H : \mathit{GoodSystem}_{M[S,C,LS,E]} \rightarrow \mathit{GoodSystem}_S$$

First of all, note that, since we have labeling functions L and $H \circ L$, H preserves labeling functions in a *strict* sense. Given variables O, C, *Atts*, D and *Conf* of sorts `Oid`, C, `AttributeSet`, `Data` and `Configuration`, respectively, we define the H function using a recursively-defined auxiliary function H' as follows:

$$H(\texttt{\{ } \mathit{Conf}\texttt{, } T \texttt{ \}}) = \texttt{\{ } H'(\mathit{Conf})\texttt{, } T \texttt{ \}}$$
$$H'(\texttt{< mon(}O\texttt{) : @Monitor | o : < } O \texttt{ : } C \texttt{ | } \mathit{Atts} \texttt{ >, data : } D \texttt{ > } \mathit{Conf})$$
$$= \texttt{< } O \texttt{ : } C \texttt{ | } \mathit{Atts} \texttt{ > } H'(\mathit{Conf})$$
$$H'(\mathit{Conf}) = \mathit{Conf} \quad \mathit{otherwise}$$

H is a function that removes all monitor objects, leaving the monitored objects as they were (without wrappers). Other objects and all messages are just left as such.

Following the methods introduced in [12], we split the rules $R_{M[S,C,LS,E]}$ into the following three disjoint sets of rules:

– Let $R^1_{M[S,C,LS,E]}$ be the set of rules without modifications, i.e., rules in R_S (rules with no objects of subclasses of C are not changed in the transformation, either if in LS or not).
– Let $R^2_{M[S,C,LS,E]}$ be the set of rules whose labels are not in LS but include objects of subclasses of C.
– Let $R^3_{M[S,C,LS,E]}$ be the set of rules whose labels are in LS and include objects of subclasses C.

Theorem 1. *H defines a (strict) simulation map from an instrumented system specification $M[S,C,LS,E]$ to a system specification S.*

Proof. Let $\rightarrow_{k,M[S,C,LS,E]}$, with $k \in \{1,2,3\}$, be the transition relation defined by $R^k_{M[S,C,LS,E]}$. We differentiate two cases:

– $a \in \mathit{GoodSystem}_{M[S,C,LS,E]}$ is rewritten to $a' \in \mathit{GoodSystem}_{M[S,C,LS,E]}$ using a rule in $R^1_{M[S,C,LS,E]}$, i.e., $a \rightarrow^1_{1,M[S,C,LS,E]} a'$. Since rules in $R^1_{M[S,C,LS,E]}$ do not have monitored objects, $H(a) = b \in \mathit{GoodSystem}_S$ can be rewritten to $H(a') = b' \in \mathit{GoodSystem}_S$ using a transition in $\rightarrow_S$.
– $a \in \mathit{GoodSystem}_{M[S,C,LS,E]}$ is rewritten to $a' \in \mathit{GoodSystem}_{M[S,C,LS,E]}$ using a rule L in $R^2_{M[S,C,LS,E]}$ or $R^3_{M[S,C,LS,E]}$, i.e. $a \rightarrow^1_{k,M[S,C,LS,E]} a'$, with $k = 2$ *or* 3. Then the rewritten subterm contains monitored objects that are removed by H. The rule in R_S from which the rule with label L was generated may then be used to rewrite $H(a) = b \in \mathit{GoodSystem}_S$ into $H(a') = b' \in \mathit{GoodSystem}_S$. □

Theorem 2. *The relation*

$$H^{-1} : \mathit{GoodSystem}_S \rightarrow \mathit{GoodSystem}_{M[S,C,LS,E]}$$

defines a (strict) simulation map from the system specification S to the instrumented system $M[S,C,LS,E]$.

Proof. H^{-1} is a relation from *valid* states in S to states in $M[S, C, LS, E]$ with monitor objects. Given a state $a \in \mathit{GoodSystem}_S$ which by $\rightarrow_S$ may be rewritten to another state $a' \in \mathit{GoodSystem}_S$. Using H^{-1}, a may be lifted to a possibly infinite number of states in $\mathit{GoodSystem}_{M[S,C,LS,E]}$. Basically, $H^{-1}(a)$ will yield states where objects of subclasses of C have been wrapped into monitor objects. All other objects and messages in the configurations will be left as such. The structure of the monitor objects introduced, including their identifier is fixed, but their `data` attribute may take any value in the `Data` sort. We prove that for all states b in $\mathit{GoodSystem}_{M[S,C,LS,E]}$ such that $H(a) = b$, a transition to a state $H(a') = b'$ in $\mathit{GoodSystem}_{M[S,C,LS,E]}$ exists in $\rightarrow_{M[S,C,LS,E]}$.

We reason by cases:

– if the state a is rewritten into a' using a rule with no objects of subclasses of C, then $H^{-1}(a) = b \in \mathit{System}_{M[S,C,LS,E]}$ and b is rewritten to some $H^{-1}(a') = b'$ using a rule in $R^1_{M[S,C,LS,E]}$.
– if the state a is rewritten by a rule whose label is not in LS but that involves objects of subclasses of C, these objects will be wrapped by monitor objects by $H^{-1}(a)$. This is the case in which a state $H^{-1} = b \in \mathit{GoodSystem}_{M[S,C,LS,E]}$ will be rewritten using a rule in $R^2_{M[S,C,LS,E]}$. In this case, there is an infinite number of possible wrappers since the variable D is free. However, since the rule label is not in LS, the value of D remains unchanged, and therefore, the state b transitions to b' so that $H(b') = a'$ in $\mathit{GoodSystem}_S$.
– if the state a is rewritten using a rule in $R^3_{M[S,C,LS,E]}$, then a can be lifted to an infinite number of possible monitor wrappers $b \in H^{-1}(a)$. Moreover, since b will transition using a monitored rule, the value of the attribute `data` matters. However, since `eval` is assumed to be a well-defined total function, for every value of D of sort *Data*, the state b can transition to a state b' such that $H(b') = a'$. □

Then, since H is a bisimulation of Kripke structures $\mathsf{A}_{M[S,C,LS,E]}$ and A_S, since strict simulations always reflect satisfaction of CTL^* formulas [12, Theorem 2], we have that given any CTL^* formula ϕ, and a configuration $a \in \mathit{GoodSystem}_{M[S,C,LS,E]}$,

$$H(a) \models_{\mathsf{A}_S} \phi \iff a \models_{\mathsf{A}_{M[S,C,LS,E]}} \phi$$

6 The throughput monitor

As an example of a *global* monitor, suppose we want to calculate the number of messages passing through nodes per time unit. By using the definition of `Data` and the `eval` function in a module extending the `MONITOR` module as shown in Figure 5, we may count the number of messages forwarded by rules per time unit.
In the module `THROUGHPUT-MONITOR`, sort `Data` is declared a supersort of 2-tuples in which the first component keeps the number of messages and the second one

```
omod THROUGHPUT-MONITOR is
  inc MONITOR .
  pr CONVERSION .

  sort 2Tuple .
  op `{_`,_`} : Nat Float -> 2Tuple [ctor] .
  subsort 2Tuple < Data .

  var  N : Nat .                            var  T : Time .
  var  Obj : Object .                       var  Thp : Float .
  vars LConf RConf Conf : Configuration .   var  Msg : Msg .

  eq eval({ N, Thp}, T, Obj, LConf, RConf)
    = if (#msgs(LConf) == #msgs(RConf))
      then { N + #msgs(LConf)),
             float(N + #msgs(LConf)) / float(T)  }
      else { N, Thp}
      fi .

  op #msgs : Configuration -> Nat .
  eq #msgs(Msg Conf) = s(#msgs(Conf)) .
  eq #msgs(Conf) = 0 [owise] .
 endom
```

Fig. 5. Throughput system monitor

the current throughput. We assume that if the number of messages in the left- and right-hand sides is the same it is because the message is being forwarded, in which case the number of messages in the `data` attribute is increased and the current number of messages is divided by the actual time.

We may rewrite the system using the `MONITOR` module expression as follows:

```
rew in MONITOR[SMP, Net, resend-msg, THROUGHPUT-MONITOR] :
  { < mon(n) : @Monitor |
       o : < n  : Net | elems : (n1 n2 n3 n4 n5 n6 n7) >,
       data : {0, 0.0} >
    < n1 : MsgCreator | targets : (n2 n3 n4 n5 n6 n7),
                        neighbors : (n2 n3 n4 n5 n6),
                        counter : 500 >
    < n2 : Node | neighbors : (n1 n3 n7) >
    < n3 : Node | neighbors : (n1 n2 n4) >
    < n4 : Node | neighbors : (n1 n3 n5) >
    < n5 : Node | neighbors : (n1 n4 n6) >
    < n6 : Node | neighbors : (n1 n5 n7) >
    < n7 : Node | neighbors : (n1 n6 n2) >, 0 } .
result GoodSystem:
  { < n1 : MsgCreator | neighbors : (n2 n3 n4 n5 n6),
                        targets : (n2 n3 n4 n5 n6 n7),
                        counter : 0 >
    < n2 : Node | neighbors : (n1 n3 n7) >
    < n3 : Node | neighbors : (n1 n2 n4) >
    < n4 : Node | neighbors : (n1 n3 n5) >
    < n5 : Node | neighbors : (n1 n4 n6) >
    < n6 : Node | neighbors : (n1 n5 n7) >
    < n7 : Node | neighbors : (n1 n6 n2) >
    < mon(n) : @Monitor |
       o : < n : Net | elems : (n1 n2 n3 n4 n5 n6 n7) >,
       data : { 3683, 2.9797734627831716 } >,
    1238}
```

The result shows, that for this execution, messages have been re-sent 3683 times, with around 2.98 messages re-sent per time unit.

7 Conclusions and future work

We have presented a methodology to define monitors that can be added to any real-time object-oriented system specification.

We have proven that the addition of these generic monitors to a system specification does *not* change its behavior. Furthermore, due to properties of simulations, safety formulas are preserved after instrumenting the specifications. This assures bisimulation by construction for any monitor and system.

Besides the theoretical results, we have presented a Maude tool which performs the *weaving* of monitors and specifications, as well as two case studies. The instrumentation has been implemented as part of Full Maude following its reflective and extensible design. We have provided a module expression that allows us to instantiate predefined generic monitors in a very simple way, perfectly integrated with Full Maude. The extended version of Full Maude, and several examples are available at `http://maude.lcc.uma.es/monitors`.

There is much work ahead. We believe that the need for indicating the rules to be monitored may be avoided when the `eval` functions have all the required information to decide when the information needs to be computed. Views from parameter monitors to specific systems may be provided, thus reducing the coupling with monitors and increasing flexibility: we may want to specify monitors depending on multiple classes or on other parameters. Multiple monitors should be used on the same systems to monitor different properties on different objects. First steps towards this kind of composition have already been taken, but the constructions will be presented elsewhere.

Acknowledgements

This work has been partially supported by Spanish MINECO/FEDER project TIN2014-52034-R, and Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech. Partially supported by NSF Grant CNS 13-19109.

References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004. http://www.labri.fr/perso/casteran/CoqArt/index.html.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. 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.
4. F. Durán. The extensibility of Maude's module algebra. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2000.

5. F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.
6. F. Durán and J. Meseguer. Maude's module algebra. *Science of Computer Programming*, 66(2):125–153, April 2007.
7. F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *J. Log. Algebr. Program.*, 81(7-8):816–850, 2012.
8. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
9. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA*, pages 279–287. CSREA Press, 1999.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. J. Meseguer. Taming distributed system complexity through formal patterns. *Sci. Comput. Program.*, 83:3–34, 2014.
12. J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. *J. Log. Algebr. Program.*, 79(2):103–143, 2010.
13. J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36. Springer, 2002.
14. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
15. C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2011.
16. J. Troya, A. Vallecillo, F. Durán, and S. Zschaler. Model-driven performance analysis of rule-based domain specific visual models. *Information & Software Technology*, 55(1):88–110, 2013.
17. S. Zschaler. Formal specification of non-functional properties of component-based software systems. *Software and System Modeling*, 9(2):161–201, 2010.