Formally Reasoning on a Reconfigurable Component-Based System - A Case Study for the Industrial World

Nuno Gaspar[1,2], Ludovic Henrio[1], and Eric Madelaine[1]

[1]Oasis Project Team,
INRIA Sophia Antipolis - Méditerranée
http://www-sop.inria.fr/oasis/

[2]ActiveEon S.A.S
http://www.activeeon.com/

{Nuno.Gaspar,Ludovic.Henrio,Eric.Madelaine}@inria.fr

Abstract. The modularity offered by component-based systems made it one of the most employed paradigms in software engineering. Precise structural specification is a key ingredient that enables their verification and consequently their reliability. This gains special relevance for *reconfigurable* component-based systems. Indeed, the ability to evolve at runtime inherently increases the complexity of an application, making its formal verification a challenging task.
To this end, the *Grid Component Model* (GCM) provides all the means to define such reconfigurable component-based applications. These, exhibit the properties of our behavioural semantics *pNets*.
In this paper we report our experience on the formal specification and verification of a reconfigurable GCM application as an industrial case study.

Keywords: Component-based Systems, Autonomous Systems, Formal Methods, Reconfiguration, Model-Checking

1 Introduction

Meeting the demands of our modern society requires special care when designing software. Applications are expected to be full-featured, performant and reliable. Moreover, for distributed applications high-availability is also cause of concern. Taming this complexity makes the use of modular techniques mandatory. To this end, the modularity offered by component-based systems made it one of the most employed paradigms in software engineering.

Embracing this approach enables structural specifications, thus leveraging formal verification. This gains special relevance for *reconfigurable* component-based systems. Indeed, while offering systems with an higher availability, the ability to evolve at runtime inherently increases the complexity of an application, making its formal verification a challenging task.

1.1 Context

This work occurs in the context of the Spinnaker Project, a French collaborative project between INRIA and several industrial partners, where we intend to contribute for the widespread adoption of RFID-based technology. To this end, our contribution comes with the design and implementation of a non-intrusive, flexible and reliable solution that can integrate itself with other already deployed systems. Specifically, we developed the HyperManager, a general purpose monitoring application with autonomic features. This was built using GCM/ProActive[1] — a Java middleware for parallel and distributed programming that follows the principles of the GCM component model. For the purposes of this project, it had the goal to monitor the E-Connectware[2] (ECW) framework in a loosely coupled manner.

For the sake of clarity let us describe one of the real life scenarios faced in a industrial context. An hotel needs to keep track of the bed sheets used by their customers. Every bed sheet used has an embedded RFID sensor chip that uniquely identifies it. At every shift, the hotel maids go through all the rooms recovering these bed sheets and putting them in a laundry cart. By reaching the end of the rooms corridor, the laundry cart emits to another physical device running the ECW Gateway software the bed sheets' identifiers. For each corridor there might be several laundry carts and one device running the ECW Gateway. After receiving the bed sheets' identifiers the ECW Gateways emit this information along with their own identifier to yet another physical device runs the ECW Server. Once the information reaches the top of this hierarchy it can be used to whatever purpose, namely bed sheets traceability.

Abstracting away this particular scenario, one can see it in a hierarchical manner as depicted by Figure 1.

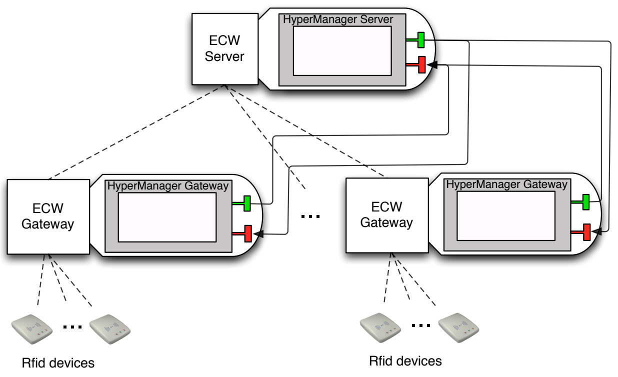


Fig. 1. Hierarchical Representation of our Case Study

[1] `http://proactive.activeeon.com/index.php`

[2] `http://www.tagsysrfid.com/Products-Services/RFID-Middleware`

Regarding our previously described scenario, this hierarchical view should pose no doubt. For each of the N floors of the hotel there are M laundry carts that communicate in a *one-to-one* style with a gateway. On the other hand, the gateways communicate with the server on a *n-to-one* style.

Moreover, the architecture depicted by Figure 1 also includes our HyperManager application. Indeed, it is deployed alongside the pre-existent distributed system, performing its monitoring on all ECW components. The careful reader will notice that the flow of requests go both from the HyperManager Server to the HyperManager Gateway, and vice-versa. Indeed, these follow the *pull* and *push* styles of communication, respectively. More details regarding these mechanisms will be discussed at a later stage.

1.2 Contributions

This paper discusses an industrial case study of a reconfigurable monitoring application. On the one hand, it should be noted that we aim at real-life applications, indeed, our models go upto the intricacies of the middleware itself. This has the direct consequence of promoting the use of formal methods within the industry.

On the other hand, we go beyond previous work [5] by including reconfiguration capabilities. This yields bigger space-states and inherently new issues to deal with. Investigating the feasibility of such undertakings is within the scope of this paper too. To the best of our knowledge this is the first work addressing the challenges of behavioural specification and verification of reconfigurable component-based applications.

1.3 Organisation of the Paper

The remaining of this paper is organised as follows. Section 2 gives the main ingredients our behavioural semantics for specifying GCM applications. Then, Section 3 presents our general purpose monitoring application - The HyperManager. Section 4 details its simplified behavioural model, i.e. without support for structural reconfigurations, and its proven properties. The impact of adding reconfiguration capabilities is discussed in Section 5. Related work is discussed in Section 6. For last, Section 7 concludes this paper.

2 *pNets* - A Behavioural Semantics for GCM Applications

This section provides a brief overview of our behavioural semantics modelling GCM/ProActive applications — *pNets*. For the sake of space, we omit some of the underlying definitions. For a detailed account of its intricacies the interested reader is pointed to [1].

As an illustrative example, the internals of a GCM primitive component featuring three service methods — m_1, m_2 and m_3 — and two client methods — m_4 and m_5 — are depicted by Figure 2.

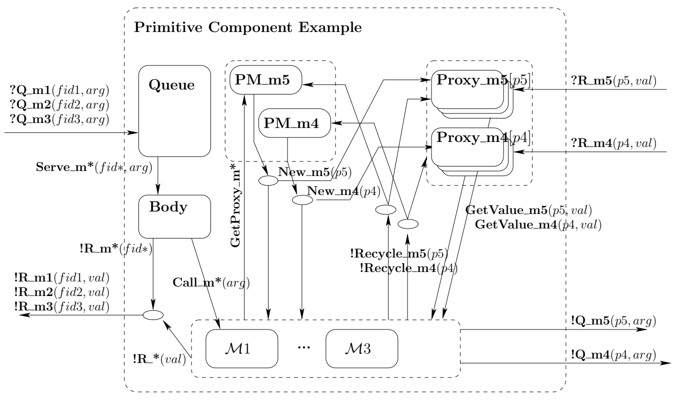


Fig. 2. pNet Representing a Primitive Component

Invocation on service methods — Q_m* — go through a Queue, that dispatches the request — Serve_m* — to the Body. Serving the request consists in performing a Call_m* to the adequate service method, represented by the $\mathcal{M}_i$ boxes in the figure. Once a result is computed, a synchronized R_m* action is emitted. This synchronization occurring between the service method and the Body stems from the fact that GCM primitive components are mono-threaded. Moreover, the careful reader will notice the $fid_{i,\ i\in\{1,2,3\}}$ in the figure. These are called *futures* and act as promises for replies, leveraging asynchrony between components.

Service methods interact with external components by means of client interfaces. This requires obtaining a proxy — GetProxy_m*, New_m$_{i,\ i\in\{4,5\}}$ — in order to be able to invoke client methods — Q_m$_{i,\ i\in\{4,5\}}$. The reply — R_m$_{i,\ i\in\{4,5\}}$ — goes to the proxy used to call the external component. Then, a GetValue_m$_{i,\ i\in\{4,5\}}$ is performed in order to access the result in the method being served. Finally, Recycle_m$_{i,\ i\in\{4,5\}}$ actions can be performed in order to release the proxies.

The behaviour of the Queue and the Body elements should pose no doubt. The former acts as priority queue with a *First in, First Out* (FIFO) policy, raising an exception if its capacity is exceeded. The latter dispatches the requests to the appropriate method and awaits its *return*, thus preventing the service of other requests in parallel.

The handling of proxies however, is not as straightforward and deserves a closer look. Figures 3 and 4 illustrate the behaviour of the Proxies and Proxy Managers, respectively. Upon reception of a New_m$_i$ action, a Proxy waits for the reply of the method invoked with it — R_m —, making thereafter its result available — GetValue_m. The proxy becomes then available on the reception of a Recycle_m action.

The behaviour of the Proxy Manager is slightly more elaborated. This maintains a *pool* of proxies, keeping track of those available and those already al-

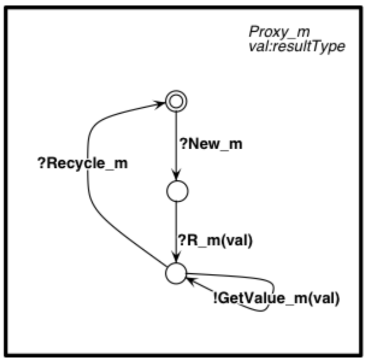


Fig. 3. Behaviour of Proxy

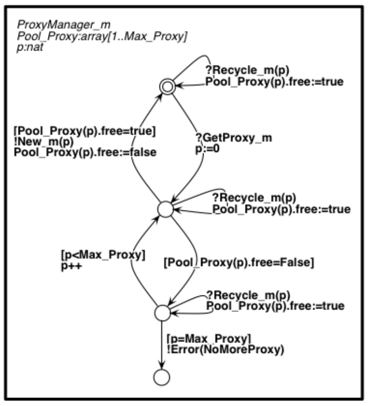


Fig. 4. Behaviour of the Proxy Manager

located. On the reception of a GetProxy_m action, it activates a new proxy — New_m — if there is one available. Should that not be the case, an Error(NoMoreProxy) action is emitted. As expected, a Recycle_m action frees a previously allocated proxy.

3 The HyperManager

The HyperManager is a general purpose monitoring application that was developed in the context of the Spinnaker Project[3]. The goal was to deliver a modular solution that would be capable of monitoring a distributed application and react to certain events. As such, the HyperManager is itself a distributed application, deployed alongside the target application to monitor.

Generally, when performing a monitoring task in an application one may consider two types of events: *pull* and *push*. The former stands for the usual communication scenario where the request comes from the client and then responded by the server. The latter however, is when the server *pushes* data to clients independently from a client's request. Both styles of communication are employed in the HyperManager application.

As illustrated by Figure 5, the server (composite) component of our monitoring application features three primitive components that are responsible for the application logic. The JMX Indicators component features only one service method: it accepts requests about a particular JMX indicator and replies its status. This encapsulates business code and interacts directly with ECW.

The Pull Component however, includes three service methods and four client interfaces. As the component's name indicates, it is responsible for *pulling* information and emitting it as *pull events*. The service methods HMStartMonitoringMethod and HMStopMonitoringMethod are responsible for starting and stopping

[3] Project OSEO ISIS. `http://www.spinnaker-rfid.com/`

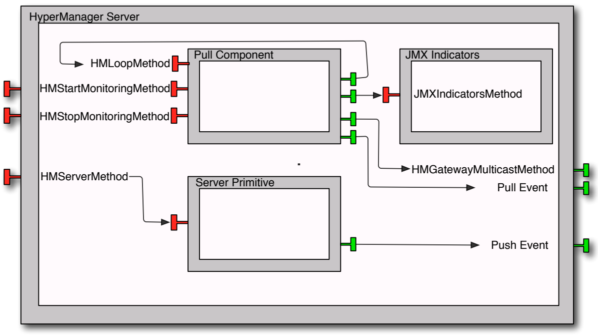


Fig. 5. HyperManager Server Component

the *pulling* activity, respectively. Typically, these are the methods called by the administrator. The remaining service method, HMLoopMethod, may pose some doubt. Indeed, it is called from one of its own client's interface. Being a ProActive application, it follows the active object paradigm where explicit threading is discouraged. As such, making a method *loop* requires this method to send itself a request before concluding its execution.

While in the monitoring loop, the HMLoopMethod method *pulls* information regarding its own local JMX indicators and those of its gateways via a *multicast*. The last remaining client interface serves the purpose of reporting the *pulled* information as *pull* events.

Last, the Server Primitive component receives *push* information from the HM Gateways — typically to alert the occurrence of some anomaly — and emits it as *push* events. In our implementation both *push* and *pull* events are then displayed in some application with graphical interface for administration purposes.

The description of the HyperManager's gateway component follow the same spirit. Figure 6 depicts its constitution.

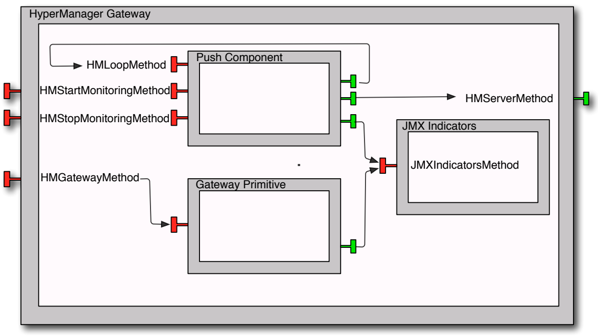


Fig. 6. HyperManager Gateway Component

It is also composed by three primitive components. As expected, the JMX Indicators component has the same semantics has described above.

The Push Component features the same service methods has the Pull Component. Its semantics however, are slightly different. While *looping* it will check for the status of its JMX indicators, and communicate with the HyperManager server if some anomaly is encountered — which will then trigger a *push* event.

As for the Gateway Primitive component, its sole purpose is to reply to the *pulling* requests from HyperManager server.

4 HyperManager's Behavioural Model

Modelling the HyperManager in our behavioural semantics *pNets* [1] requires us to provide a behaviour for each service method. In the following we illustrate this by providing an *user-version* LTS for all of them — i.e. we omit all the machinery involving futures and proxies. Moreover, for more material on this case study the reader is invited to its companion website[4].

Regarding our modelling and verification *workflow*, we build our behavioural models by encoding the involved processes in the Fiacre specification language [3]. Then, the flac compiler translates it to Lotos [4]. From there we can use the CADP toolbox [7]. Typically, we use bcg_open for space-state generation — in conjunction with distributor if performing it on a distributed setting —, svl scripts for managing space-state replication, label renaming and build products of transition systems. For last, evaluator4 for model-checking our space-state against MCL (Model Checking Language) [10] formulas — an extension of the alternation-free regular μ-calculus with facilities for manipulating data.

To optimize the size of the model, the composite components have no request queue and requests are directly forwarded to the targeted primitive component. This has no influence in the system's semantics as the primitives' request queues are sufficient for dealing with asynchrony and requests from the sub-components are directly dispatched too. Moreover, we set the primitive components with re-entrant calls with a queue of size 2, and the remaining of size 1.

4.1 The HM Gateway

The JMX Indicators primitive component only features one service method: JMXIndicatorsMethod. Its behaviour is modelled by Figure 7. For the sake of simplicity, we only model two types of indicators: *MemoryUsage* and *DeviceStatus*. The latter takes into account an identifier, returning its availability status. This relates to the status of a RFID reader transmitting to the ECW Gateway. While the former simply returns the stability status of the memory.

The service method offered by the Gateway Primitive component has also a fairly simple behaviour. It is illustrated by Figure 8. It acts merely has a request forwarder for the JMX Indicators component.

[4] http://www-sop.inria.fr/members/Nuno.Gaspar/HyperManager.php

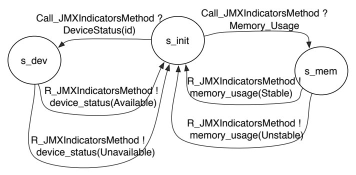

Call_JMXIndicatorsMethod ? DeviceStatus(id)
Call_JMXIndicatorsMethod ? Memory_Usage
s_dev
s_init
s_mem
R_JMXIndicatorsMethod ! device_status(Available)
R_JMXIndicatorsMethod ! device_status(Unavailable)
R_JMXIndicatorsMethod ! memory_usage(Stable)
R_JMXIndicatorsMethod ! memory_usage(Unstable)

Fig. 7. Behaviour of the JMXIndicatorsMethod

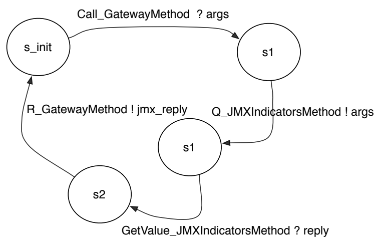

Call_GatewayMethod ? args
s_init
s1
R_GatewayMethod ! jmx_reply
Q_JMXIndicatorsMethod ! args
s1
s2
GetValue_JMXIndicatorsMethod ? reply

Fig. 8. Behaviour of the HMGatewayMethod

Regarding the Push Component, the HMStartMethod and HMStopMethod methods enable/disable the *looping* process. This is achieved by a shared variable among processes that acts as a *flag*. Invoking HMStartMethod will set our *flag* variable **started** to *true* and perform an invocation to HMLoopMethod. On the other hand, HMStopMethod will set the *flag* to *false*. Their behaviour is rather trivial and therefore omitted for the sake of space. In practice, the involved labels are GuardQuery, GuardReply?b:bool, SetFalse and SetTrue; their meaning should be obvious from their names.

The last remaining service method to describe is the most interesting one — the *loop* method.

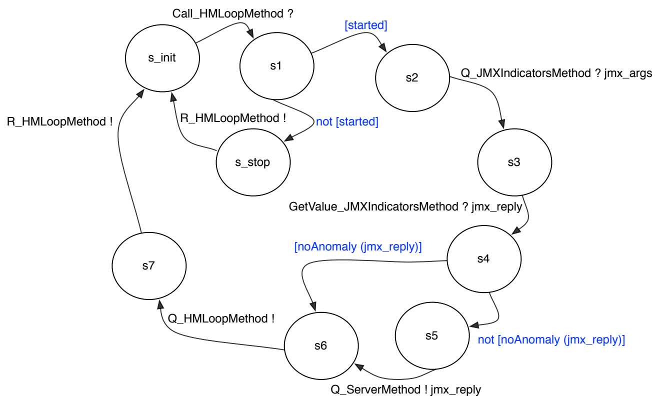


Fig. 9. Behaviour of the HMLoopMethod at the Gateway level

As illustrated by Figure 9, the actual *looping* only occurs if our *flag* variable is set to TRUE, otherwise we just *return* without performing any significant action. While *looping*, we check our JMX indicators. Should an anomaly be detected we report it to the HM Server. Last, before *returning* we send a request to ourselves — Q_HMLoopMethod — in order to be able to continue *looping* while our *flag* variable evaluates to *true*.

Model Generation and Proven Properties Table 1 illustrates the relevant information concerning Gateway's space state built using the CADP toolbox.

	States	Transitions	File Size
hmgateway.bcg	14.931.628	147.485.103	∼ 295 mb
hmgateway-min.bcg	14.931.628	147.485.103	∼ 296 mb

Table 1. Numbers Regarding the Gateway Model

The entry suffixed by -min mean that minimization by branching *bissimulation* was applied. We note that the minimization process fails to produce a reduced transition system. However, there is an increase in the file size even though the number of states and transitions remained equal. This can be justified by the fact that bcg_min inserts information in the produced file stating that it came from a minimization process. In any case, this overhead is rather negligible.

Having this *space-state* generated we can now prove some properties regarding the expected behaviour of our model. For instance, one could wonder about this rather unusual *looping* mechanism. Once setting our *flag* to *true*, we continue *looping* until we receive a request to stop monitoring.

Property 1.
```
[ "Q_HMSTARTMETHOD" . "Q_HMLOOPMETHOD" .
  (not "Q_HMSTOPMETHOD")* . "GUARDREPLY !FALSE" ] false
```

Naturally, we also want to avoid overloading the HM Server with unnecessary messages. As such, we want to ensure that we cannot *push* data if not in the presence of an anomaly. This can be modelled as follows:

Property 2.
```
[ ((not "R_JMXINDICATORS_ToPush !MEMORY_USAGE (UNSTABLE)")* .
      "Q_SERVERMETHOD.*")  |
  ((not "R_JMXINDICATORS_ToPush !DEVICE_STATUS ((UNAVAILABLE, IDTWO))")* .
      "Q_SERVERMETHOD.*") |
  ((not "R_JMXINDICATORS_ToPush !DEVICE_STATUS ((UNAVAILABLE, IDONE))")* .
      "Q_SERVERMETHOD.*")
] false
```

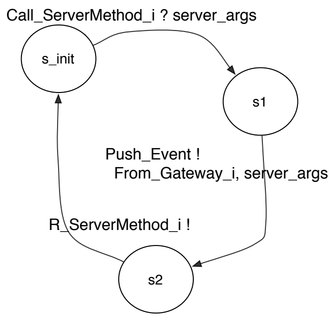
Both properties are naturally proved *true*.

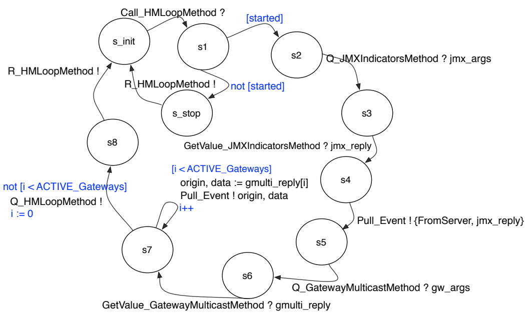
4.2 The HM Server

Similarly as seen for the HM Gateway component, our HM Server component also features a JMX Indicators primitive component. This however, is naturally not endowed with indicators for the RFID devices statuses. Technically, we attach to the LTS modelling its behaviour (Figure 7) a context that constraints its requests. Moreover, HMStartMethod and HMStopMethod methods exhibit the same behaviour as described above.

As seen above, upon detection of an anomaly, the HM Gateway component *push*es the relevant information to the HM Server. Then, it is emitted as a *push* event as depicted by Figure 10. The careful reader will notice that the emitted event also contains the information regarding the HM Gateway from which the anomaly originated. This should come as no surprise as there can be several of them, and properly identifying the source of an abnormal situation is of paramount importance.

As depicted by Figure 11, the *looping* process for the HM Server proceeds in a similar fashion as the one from the HM Gateway: the *flag* variable **started**'s valuation determines whether we enter the *looping* process or if we just *return*. While *looping* we *pull* information from the local JMX indicators and emit it as a *pull* event.


Call_ServerMethod_i ? server_args
s_init
s1
Push_Event !
From_Gateway_i, server_args
R_ServerMethod_i !
s2
Fig. 10. Behaviour of the HM-ServerMethod


Call_HMLoopMethod ?
[started]
s_init
s1
s2
Q_JMXIndicatorsMethod ? jmx_args
R_HMLoopMethod !
R_HMLoopMethod !
not [started]
s_stop
s3
s8
GetValue_JMXIndicatorsMethod ? jmx_reply
[i < ACTIVE_Gateways]
origin, data := gmulti_reply[i]
Pull_Event ! origin, data
i++
s4
not [i < ACTIVE_Gateways]
Q_HMLoopMethod !
i := 0
Pull_Event ! {FromServer, jmx_reply}
s5
s7
s6
Q_GatewayMulticastMethod ? gw_args
GetValue_GatewayMulticastMethod ? gmulti_reply
Fig. 11. Behaviour of the HMLoopMethod at the Server level

Moreover, via a multicast interface we also *pull* information from the bound gateways. This, will make us emit as many *pull* events as the number of bound gateways. Last, we perform a request to ourselves in order to continue *looping*.

Model Generation and Proven Properties Table 2 illustrates the relevant information concerning HM Server's space-state.

	States	Transitions	File Size
hmserver.bcg	12.787.376	187.589.422	∼ 363 mb
hmserver-min.bcg	12.787.376	187.589.422	∼ 396 mb

Table 2. Numbers Regarding the Server Model

As in the case of our HM Gateway model, the minimization process failed to produce a smaller space-state. However, this time we get a 9% increase in the file size, not so much negligible as the increase noticed for the HM Gateway's space-state.[5]

A rather trivial property we can expect to hold is that we can reach a state which *explodes* one of the request queues. This can be modelled in MCL as follows:

Property 3. `< true* . 'QUEUEEXCEPTION_SERVERPRIMITIVE !.*'> true`

As mentioned above, we omitted all the machinery involving proxies while describing the service methods' behaviour. However, this is naturally included in our generated model. For instance, the HMLoopMethod method needs to request a proxy in order to be able to invoke JMX Indicators's service method. This is naturally encoded as follows:

Property 4. `[ (not "GETPROXY_JMXINDICATORSMETHOD.*")* .`
`"Q_JMXINDICATORSMETHOD_FromPullComponent.*"] false`

As expected, both properties hold in our model.

4.3 System Product, Model Generation and Proven Properties

We attempted to generate a system product constituted by two HM Gateways and one HM Server components. However, even on a machine with 90 GB of RAM, we experienced the so common space-state explosion phenomena.

This arises often in the analysis of complex systems. To this end, *communication hiding* comes as an efficient and pragmatic approach for tackling this issue. Indeed, it allows to specify the *communication actions* that need not to be observed for verification purposes, thus yielding more tractable space-states.

Table 3 illustrates the effects of applying this technique to our model. The sole *communication actions* being hidden are the ones involved in (1) the request transmission from the Queue to the adequate method — Serve_ and Call_ —, (2) the proxy machinery —GetProxy_, New_ and Recycle_ —, and (3) finally in the *guard* of the *looping* methods — GuardQuery, GuardReply, SetFalse and SetTrue.

The lines suffixed by -hidden indicate the results obtained by *hiding* the mentioned *communication actions* in the *minimized* HM Gateway and HM server space-states. For both, no effect is noticed on the size of the LTS. However, there is a decrease in the file size. This is due to the fact that the *hiding* process yields several τ-transitions, which facilitates file compression. This has the consequence of leveraging the posteriori *minimization* process. Indeed, we even obtain a reduction by two orders of magnitude (!) for the HM Gateway space-state.

The HyperManager comes as a monitoring application that should be able to properly trace the origin of an anomaly. As such, one behavioural property

[5] In fact, we encountered another peculiar situation where minimization produced a smaller space-state, yet a bigger file size: `http://cadp.forumotion.com/t374-bcg-file-size-after-minimization`

	States	**Transitions**	**File Size**
hmgateway-min-w-hidden.bcg	14.931.628	147.485.103	∼ 287 mb
hmgateway-min-w-hidden-min.bcg	409.374	4.007.232	∼ 8.5 mb
hmserver-min-w-hidden.bcg	12.787.376	187.589.422	∼ 375 mb
hmserver-min-w-hidden-min.bcg	5.761.504	85.157.420	∼ 179 mb
SystemProduct.bcg	342.047.684	3.026.114.393	∼ 5.27 gb
SystemProduct-min.bcg	259.340.044	2.396.896.830	∼ 4.83 gb

Table 3. Relevant numbers regarding our generated model

that we expect to hold is that whenever an abnormal situation is detected by a HM Gateway, it is *fairly inevitable* to be reported as a *push event* that correctly identifies its origin.

First, we shall use MCL's macros capabilities to help us build our formula:

```
macro GETVALUE_1_MEMORY () =
  "GETVALUE_JMXINDICATORSMETHOD_Push_1 !MEMORY_USAGE (UNSTABLE)"
end_macro

macro PUSH_1_MEMORY ()     =
  ("PUSH_EVENT !PUSH_EVENT (UNSTABLEMEMORYUSAGE, FIRSTGATEWAY)")
end_macro
...
```

The above macros should be self-explanatory. The former represents the detection of an anomaly coming from the first HM Gateway — the model is instantiated with two HM Gateways, thus we differentiate their actions by suffixing them adequately. The latter stands for the emission of the *push* event corresponding to that anomaly. The macros for the remaining relevant actions are defined analogously.

Moreover, we define the following macro generically encoding the *fair inevitability* that after an *anomaly* the system emits a *push.*

```
macro FAIRLY_INEVITABLY_A_PUSH (ANOMALY, PUSH) =
       [ true* . "ANOMALY" . (not "PUSH")* ]
           < (not PUSH)* . PUSH > true
end_macro
```

Having our macros defined, we can now write our formula of interest:

Property 5.

```
(FAIRLY_INEVITABLY_A_PUSH(GETVALUE_1_MEMORY, PUSH_1_MEMORY) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_2_MEMORY, PUSH_2_MEMORY) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_1_DEVICE_1, PUSH_1_DEVICE_1) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_1_DEVICE_2, PUSH_1_DEVICE_2) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_2_DEVICE_1, PUSH_2_DEVICE_1) and
 FAIRLY_INEVITABLY_A_PUSH(GETVALUE_2_DEVICE_2, PUSH_2_DEVICE_2)
)
```

As expected, this property holds for our model.

5 The Case Study Reloaded: On Structural Reconfigurations

As seen so far, the HyperManager acts as a monitoring application with two styles of communication: *pull* and *push*. However, it also needs to cope with structural reconfigurations. This means that at runtime the architecture of the application can evolve by, say, establishing new bindings and/or removing existing ones.

For GCM applications *bind* and *unbind* operations are handled by the component owning the *client* interface that is supposed to be reconfigurable. This should come as no surprise, indeed, it follows the same spirit as in object-oriented languages: an object holds the reference to a target object; it is this object that must change the reference it holds.

In our case-study, these reconfigurations can occur both at the server level — when *pulling* data from the bound gateways —, and at gateway level — when *pushing* data to the server. The difference lies at the fact that the server communicates via a *multicast* interface, unlike the gateways that establish a standard *1-to-1* communication. Therefore, these are dealt in a different manner.

5.1 HM Reconfigurable Gateway

Let us first illustrate how a *singleton client* reconfigurable interface is modelled in *pNets*. As depicted by Figure 12, for each client reconfigurable interface there exists a *binding controller*.

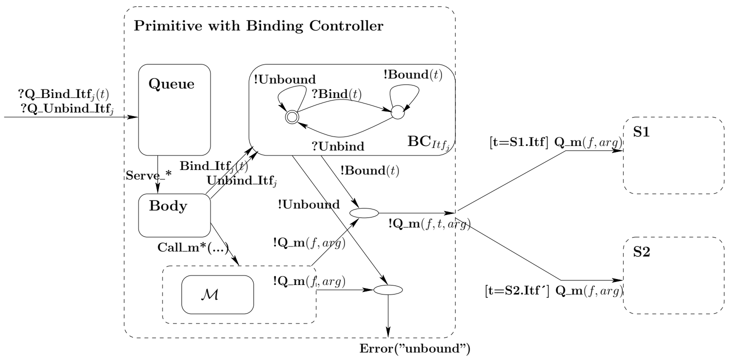


Fig. 12. Binding Controller

Indeed, we allow for reconfigurations by defining two new request messages for the *binding* and *unbinding* of interfaces. These are delegated to a binding controller that upon method invocation over these reconfigurable interfaces will check if they are indeed bound, emitting an error if it is not the case. Moreover, the target of the invocation is decided by checking its passed reference. For this reason one must know statically what are the possible target interfaces that a reconfigurable interface can be bound too.

In practice, to our HM Gateway model discussed in Subsection 4.1 we add the request messages Q_BIND_SERVERMETHOD and Q_UNBIND_SERVERMETHOD. Since we only have one reconfigurable interface we can avoid adding an explicit parameter — unlike shown in Figure 12, where we demonstrate a more general case. Moreover, since the gateways can only be bound to one target — the server — the *binding controller* only needs to keep a state variable regarding its *boundedness*.

As expected, these changes have a considerable impact in the size of our model. This is illustrated by Table 4.

	States	**Transitions**	**File Size**
hmgateway-reconfig.bcg	354.252.868	4.178.400.886	∼ 8.45 gb
hmgateway-reconfig-min.bcg	354.104.012	4.176.956.686	∼ 8.54 gb

Table 4. Gateway with Reconfigurable Interface

All the properties proven in Subsection 4.1 still hold for this new HM Gateway model, with a natural overhead in *model-checking* them in a much bigger space-state. However, for this new model we are more interested in addressing the reconfiguration capabilities. For instance, provided that our interface is bound, it will not yield an UNBOUND action upon method invocation.

Property 6.
```
< true* . "Q_BIND_SERVERMETHOD" . (not "Q_UNBIND_SERVERMETHOD")*  .
"Q_SERVERMETHOD"  .  (not "Q_UNBIND_SERVERMETHOD")* . "UNBOUND"  > true
```

The above property is proved *false*, indicating that indeed a path in our space-state yielding an UNBOUND action despite the interface being bound will not occur.

5.2 HM Reconfigurable Server

As an illustrative example, the *pNet* of a primitive component featuring a reconfigurable client *multicast* interface and two service methods — m_1 and m_2 — is depicted by Figure 13.
In short, the machinery involved for dealing with this kind of interfaces mainly differs from reconfigurable *singleton* interfaces in that we must keep track of the target's statuses boundedness. Indeed, the emission of a new proxy — $\mathsf{New\_m}_{i,i\in\{1,2\}}$ — is synchronized in a similar manner, however we also transmit the current status of the *multicast* interface (i.e. the G variable in the figure). This status will be taken into account when invoking one of the client methods — $\mathsf{Q\_m}_{i,i\in\{1,2\}}$. In practice, G is a boolean vector whose element's valuation determine the interface's boundedness.

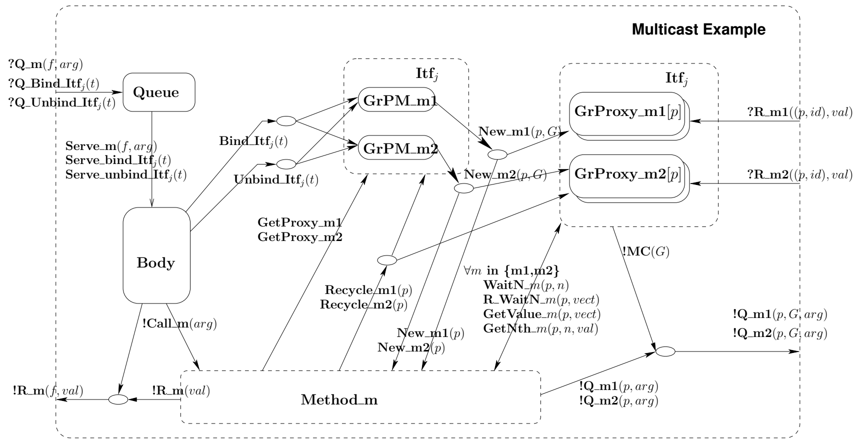


Fig. 13. pNet Example for Reconfigurable Multicast Interface

Table 5 demonstrates the impact of adding reconfiguration capabilities to our HM Server model.

	States	**Transitions**	**File Size**
hmserver-reconfig.bcg	931.640.080	16.435.355.306	∼ 32.93 gb

Table 5. Server with Reconfigurable *Multicast* Interface

The generated space-state for our HM Server model nearly attained 1 billion states.[6] Our attempts to *minimize* it revealed to be unsuccessful due to the lack of memory. These were carried out on a workstation with ∼90 GB of RAM.

It is worth noticing that while we were not able to *minimize* the produced space-state, we were still able to *model-check* it against the same properties discussed in Subsection 4.2.

5.3 Model Generation and Proven Properties

As seen in Subsection 4.3, building the product of our system already showed to be delicate. Abstraction techniques such as *communication hiding* were already

[6] As mentioned in Subsection 4.2, for our HM Server model, the JMX Indicators component is generated with a context not contemplating the request of device statuses. Previous experiments not considering this context produced a HM Server model with the following characteristics: 4.148.563.680 states, with 74.268.977.628 transitions, on a 154.2 GB file. It is interesting to note the huge impact that (the lack of) a contextual space-state generation on one of its components can provoke.

required to build our system. Thus, it should come as no surprise that we face the same situation here.

However, it should be noted that the *hiding* process itself, produced little effect on the file size, and no effect on the space-states. It mainly acted as a means to leverage a posteriori *minimization* process, allowing for a very significant space-state reduction. Table 6 illustrates the results obtained by following the same approach as above.

	States	**Transitions**	**File Size**
hmgateway-reconfig-min-w-hidden.bcg	354.104.012	4.176.956.686	~ 8.15 gb
hmgateway-reconfig-min-w-hidden-min.bcg	11.090.974	127.799.874	~ 283.5 mb
hmserver-reconfig-min-w-hidden.bcg	931.640.080	16.435.355.306	~ 31.28 gb

Table 6. Relevant Numbers Regarding our Generated Model with Reconfigurable Interfaces

We obtained a significant space-state reduction for the HM Gateway model, but we were unable to *minimize* the HM Server. Indeed, *communication hiding* may leverage space-state reduction, but still requires that the *minimization* process is able to run, therefore not solving the lack of memory issue. This is a rather embarrassing situation as we would expect a significant space-state reduction as well for the HM Server.

While *communication hiding* revealed to be a valuable tool, *minimization* is still a bottleneck if the input space-state is already too big. Thus, we need to shift this burden to the lower levels of the hierarchy. Indeed, both HM Server and HM Gateway components are the result of a product between their primitive components. Moreover, these are themselves the result of a product between their internals – request queue, body, proxies ...

Table 7 illustrates the results obtained by hiding the same communication actions as in the above approaches, but before starting to build any product.

	States	**Transitions**	**File Size**
hidden-hmgateway-reconfig.bcg	3.483.000	43.193.346	~ 85.46 mb
hidden-hmgateway-reconfig-min.bcg	3.073.108	39.373.968	~ 83.95 mb
hidden-hmserver-reconfig.bcg	210.121.904	3.890.791.694	~ 7.52 gb
hidden-hmserver-reconfig-min.bcg	177.604.848	3.288.937.718	~ 6.61 gb
SystemProduct-reconfig.bcg	539.979.906	6.041.011.217	~ 11.44 gb

Table 7. Relevant Numbers Regarding our Generated Model with Reconfigurable Interfaces

Indeed, following this approach proved to be fruitful as we were able to generate the system product. Yet, *minimization* remained still out of reach.

Nevertheless, we are still in a position to *model-check* some properties of interest. For instance, *pulling* information via a *multicast* emission is now predicated with a boolean array whose element's valuation determines its boundedness. As an example, a rather simple *liveness* property is the following one:

Property 7.

```
<true* . "Q_GATEWAYMULTICASTMETHOD !ARRAY(FALSE FALSE) !MEMORYUSAGE"> true
```

Initially, both HM Gateways are bound, the above property tell us that we can indeed unbind both of them.

6 Related Work

Many works can be found in the literature embracing a behavioural semantics approach for the specification and verification of distributed systems. Yet, literature addressing the aspects of reconfigurable applications remains scarce. Nevertheless, we must cite the work around BIP (Behaviour, Interaction, Priority) [2] — a framework encompassing rigorous design principles. It allows the description of the coordination between components in a layered way. Moreover, it has the particularity of also permitting the generation of code from its models. Yet, structural reconfigurations are not supported.

Another rather different approach that we must refer is the one followed by tools specifically tailored for architectural specifications. For instance, in [9] Inverardi et. al. discusses CHARMY, a framework for designing and validating architectural specifications. It offers a full featured graphical interface with the goal of being more *user friendly* in an industrial context. Still, architectural specifications remain of static nature.

Looking at the interactive theorem proving arena we can also find some related material. In [6] Boyer et. al. propose a reconfiguration protocol and prove its correctness in The Coq Proof Assistant. This work however, focuses on the protocol itself, and not in the behaviour of a reconfigurable application. Moreover, in [8] we presented Mefresa — a ***Me****chanized* ***F****ramework for* ***Re****asoning on* ***S****oftware* ***A****rchitectures*. This discusses a (re)configuration language and its underlying formal semantics for reasoning at the architectural level.

7 Final Remarks

In the realm of component-based systems, behavioural specification is among the most employed approaches for the rigorous design of applications. It leverages the use of Model-Checking techniques, by far the most widespread formal method in the industry. Yet, verification in the presence of structural reconfigurations remains still as a rather unaddressed topic. This can be justified by the inherent complexity that such systems impose. However, playing a significant role for the increase in systems availability, and key ingredient in the autonomic computing arena, tackling its demands should be seen of paramount importance.

In this paper we discussed the specification and formal verification of a reconfigurable monitoring application as an industrial case-study. Modelling our HyperManager application upto the intricacies of the middleware lead us to a combinatorial explosion in the set of states. This, is further aggravated by the inclusion of reconfigurable interfaces. Even the use of compositional and contextual space-state generation techniques revealed to be insufficient. While this could be solved by further increasing the available memory in our workstation, it is worth noticing that this approach is not always feasible in practice. As usual in the realm of formal verification, abstraction is the key. Taking advantage of CADP's facilities for *communication hiding*, one can specify actions that need not to be observed for our verification purposes, which further enhances the effects of posteriori minimization by branching *bissimulation*. This illustrates the pragmatic rationale of formal verification by Model-Checking — the most likely reason behind its broad acceptance in the industry.

References

1. Rabéa Ameur-Boulifa, Ludovic Henrio, Eric Madelaine, and Alexandra Savu. Behavioural Semantics for Asynchronous Components. RR RR-8167, December 2012.
2. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3):41–48, 2011.
3. B. Berthomieu, J.P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The syntax and semantics of FIACRE. RR, 2009.
4. Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, March 1987.
5. Rabéa Ameur Boulifa, Raluca Halalai, Ludovic Henrio, and Eric Madelaine. Verifying safety of fault-tolerant distributed components. In *International Symposium on Formal Aspects of Component Software (FACS 2011)*, 2011.
6. Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust reconfigurations of component assemblies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13. IEEE Press, 2013.
7. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2011.*
8. Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. Bringing Coq Into the World of GCM Distributed Applications. *International Journal of Parallel Programming*, 2013. HLPP'2013 Special Issue.
9. P. Inverardi, H. Muccini, and P. Pelliccione. Charmy: An extensible tool for architectural analysis. In *ESEC-FSE'05, ACM SIGSOFT Symposium on the Foundations of Software Engineering. Research Tool Demos*, September 5-9, 2005.
10. Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 148–164, Berlin, Heidelberg, 2008. Springer-Verlag.