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

Nuno Gaspar[1,2,3](✉), Ludovic Henrio[2], and Eric Madelaine[1,2]

[1] INRIA Sophia Antipolis, Valbonne, France
{Nuno.Gaspar,Eric.Madelaine}@inria.fr
[2] University of Nice Sophia Antipolis,
CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
Ludovic.Henrio@cnrs.fr
[3] ActiveEon S.A.S, Sophia Antipolis, France
http://www.activeeon.com/

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.

To this end, the *Grid Component Model* (GCM) provides all the means to define such reconfigurable component-based applications. 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.

J.L. Fiadeiro et al. (Eds.): FACS 2013, LNCS 8348, pp. 137–156, 2014.
DOI: 10.1007/978-3-319-07602-7_10,

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 running 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 Fig. 1.

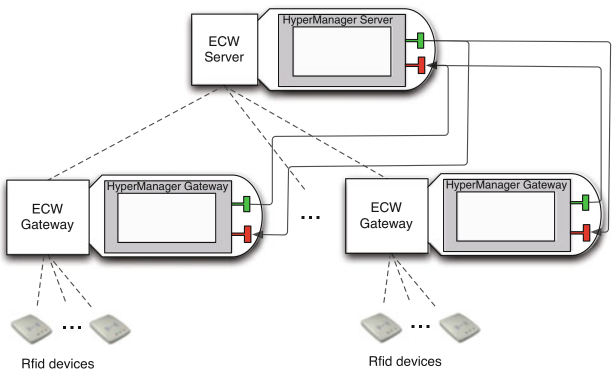

Fig. 1. Hierarchical representation of our case study

1 http://proactive.activeeon.com/index.php
2 http://www.tagsysrfid.com/Products-Services/RFID-Middleware

Regarding the 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, there is also the need to cope for possible maintenance issues. For instance, in the case of malfunction of some device running the ECW Gateway, it may be required to replace it or add a new one in order to avoid any overloading.

The architecture depicted by Fig. 1 also includes the 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 state-spaces 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 of our behavioural semantics for specifying GCM applications. Then, Sect. 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 Sect. 5. Related work is discussed in Sect. 6. For last, Sect. 7 concludes this paper.

2 A Behavioural Semantics for GCM Applications

This section provides a brief overview of the behavioural semantics modelling GCM/ProActive applications by relying on the *pNets* formalism. 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].

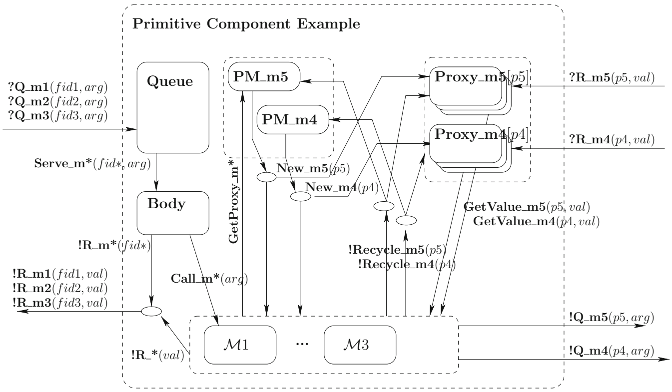

Fig. 2. pNet representing a primitive component

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 Fig. 2.

Invocation on service methods — $\mathsf{Q\_m}_{i,\ i\in\{1,2,3\}}$ — 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*, $\mathsf{New\_m}_{i,\ i\in\{4,5\}}$ — in order to be able to invoke client methods — $\mathsf{Q\_m}_{i,\ i\in\{4,5\}}$. The reply — $\mathsf{R\_m}_{i,\ i\in\{4,5\}}$ — goes to the proxy used to call the external component. Then, a $\mathsf{GetValue\_m}_{i,\ i\in\{4,5\}}$ is performed in order to access the result in the method being served. Finally, $\mathsf{Recycle\_m}_{i,\ i\in\{4,5\}}$ 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 $\mathsf{New\_m}_i$ action, a Proxy waits for the reply of the method invoked with it — R_m —, making thereafter its result

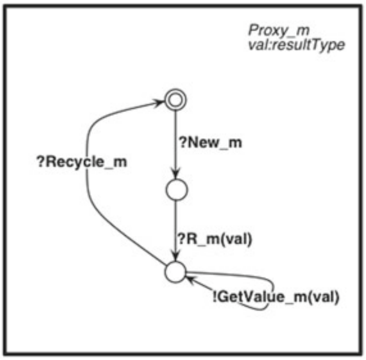

Fig. 3. Behaviour of proxy

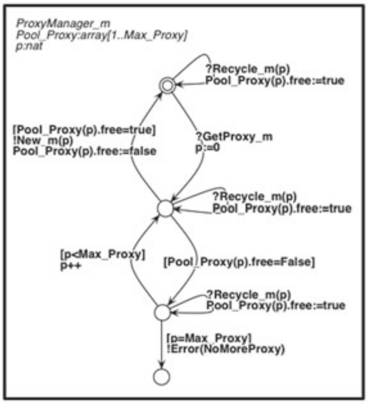

Fig. 4. Behaviour of the proxy manager

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 allocated. 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 Fig. 5, the server (composite) component of the HyperManager application features three primitive components that are responsible for the application logic. Each possesses one or several service methods that

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

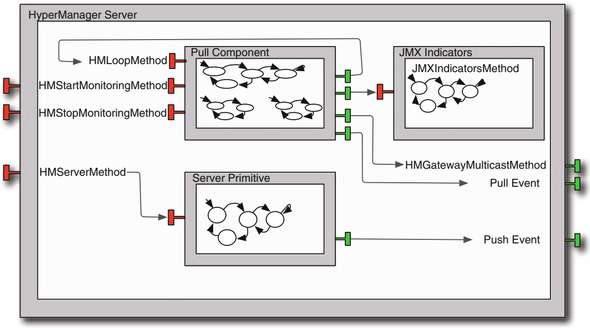

Fig. 5. HyperManager server component

stand for their functionalities and are modelled by *labelled transition systems* (LTSs).

The JMX Indicators component features only one service method: it accepts requests about a particular JMX[4] 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 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* is achieved by making this method sending 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* client interface. 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 a graphical interface for administration purposes.

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

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

The Push Component features the same service methods as the Pull Component. Its semantics however, are slightly different. While *looping* it will check for

[4] JMX is the standard protocol used for monitoring Java applications.

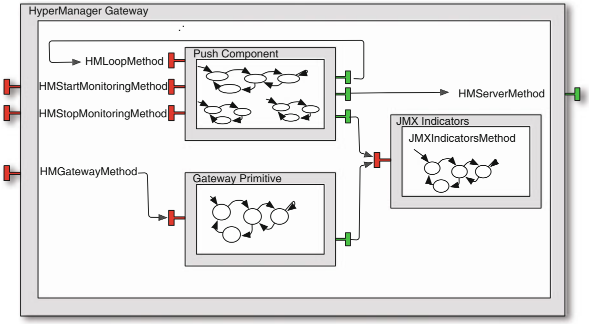

Fig. 6. HyperManager gateway component

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 the 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[5].

Regarding our modelling and verification *workflow*, we build the 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 [9]. Typically, we use bcg_open for state-space generation — in conjunction with distributor if performing it on a distributed setting —, svl scripts for managing state-space replication, label renaming and build products of transition systems. For last, evaluator4 for model-checking the state-space against MCL (Model Checking Language) [13] 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.

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

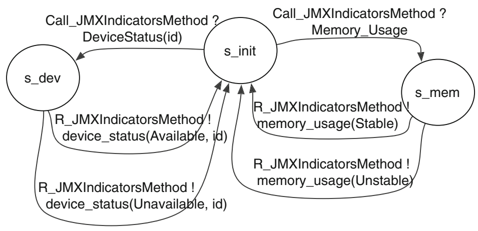

Fig. 7. Behaviour of the JMXIndicatorsMethod

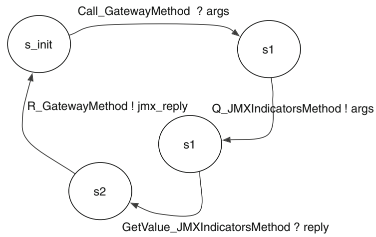

Fig. 8. Behaviour of the HMGatewayMethod

4.1 The HM Gateway

The JMX Indicators primitive component only features one service method: JMXIndicatorsMethod. Its behaviour is modelled by Fig. 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 Fig. 8. It acts merely as a request forwarder for the JMX Indicators component.

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 the *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.

As illustrated by Fig. 9, the actual *looping* only occurs if the *flag* variable **started** is set to TRUE, otherwise a simple *return* without performing any significant action is made.[6] While *looping*, the JMX indicators are checked. Should an anomaly be detected a report is made to the HM Server. Last, before *returning* a request is sent to itself — Q_HMLoopMethod — in order to be able to continue *looping* while the *flag* variable evaluates to *true*.

Model Generation and Proven Properties. Table 1 illustrates the relevant information concerning the HM Gateway's state-space built using the CADP toolbox. The model is generated with two RFID readers.

[6] For the sake of clarity, *communications actions* are written in black, while *local* computations are written in blue. Their intended meaning should pose no doubt.

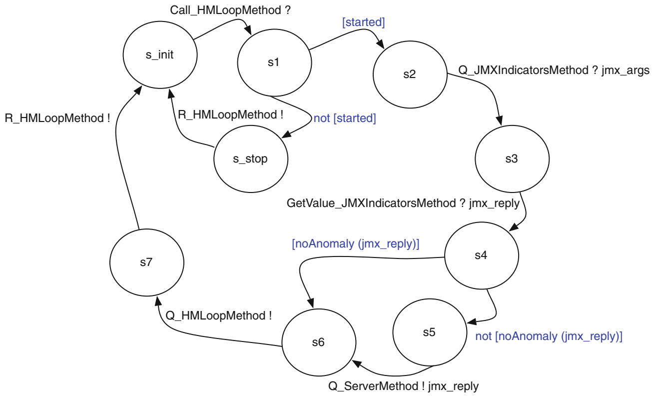

Fig. 9. Behaviour of the HMLoopMethod at the gateway level

Table 1. Numbers regarding the gateway model

	States	Transitions	File size (mb)
hmgateway.bcg	14.931.628	147.485.103	∼295
hmgateway-min.bcg	14.931.628	147.485.103	∼296

The entry suffixed by -min means that minimization by branching *bisimulation* was applied. We note that the minimization process fails to produce a reduced transition system. This is due to the fact that we do not hide any *communication action* and all transitions are visible[7]. However, there is an increase in the file size even though the number of states and transitions remained equal. This is 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 state-space generated we can now prove some properties regarding the expected behaviour of the model. Specifying properties of interest in MCL is a rather intuitive task due to its expressiveness and conciseness. Its main ingredients include patterns extracting data values from LTS actions, modalities on transition sequences described using extended regular expressions and programming language constructs.

For instance, one could wonder about this rather unusual *looping* mechanism. Once setting the *flag* to *true* — accomplished by Q_HMStartMethod —, the *looping* continues until a request to stop monitoring is received. That is, there is no path in which the *flag* evaluates to *false* without the occurrence of a Q_HMStopMethod.

[7] It is worth noticing that the FLAC compiler translates shared variables and internal communications into τ-transitions. These will therefore disappear from the LTS if subject to minimization.

Property 1. `[ "Q_HMStartMethod" . "Q_HMLoopMethod" . (not"Q_HMLoopMethod")* . "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_JMXIndicatorsMethod !memory_usage (Unstable)")* .
      "Q_ServerMethod.*")  |
  ((not "R_JMXIndicatorsMethod !device_status ((Unavailable, IdTwo))")* .
      "Q_ServerMethod.*") |
  ((not "R_JMXIndicatorsMethod !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, the 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 (Fig. 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 Fig. 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 Fig. 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.

Moreover, via a multicast interface information is *pulled* from the connected gateways. This, will emit as many *pull* events as the number of connected gateways. Last, a request to itself is performed in order to continue *looping*.

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

As in the case of the HM Gateway model, the minimization process failed to produce a smaller state-space. However, this time we get a 9 % increase in the

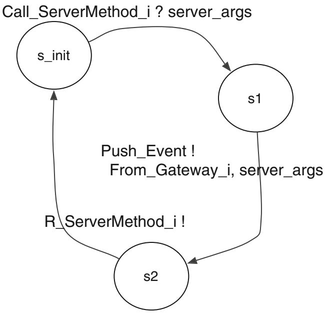

Fig. 10. Behaviour of the HMServerMethod

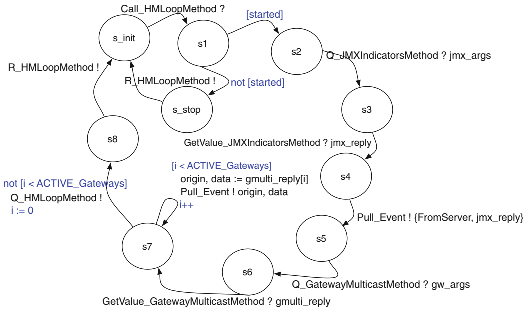

Fig. 11. Behaviour of the HMLoopMethod at the server level

Table 2. Numbers regarding the server model

	States	Transitions	File size (mb)
hmserver.bcg	12.787.376	187.589.422	∼363
hmserver-min.bcg	12.787.376	187.589.422	∼396

file size, not so much negligible as the increase noticed for the HM Gateway's state-space.[8]

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 the 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.*"] false
```

As expected, both properties hold in the 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 state-space explosion phenomena.

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

Table 3. Relevant numbers regarding the generated model

	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

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 state-spaces.

Table 3 illustrates the effects of applying this technique to the 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 state-spaces. 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 subsequent *minimization* process. Indeed, we even obtain a reduction by two orders of magnitude (!) for the HM Gateway state-space.

The HyperManager comes as a monitoring application that should be able to properly trace the origin of an anomaly. As such, one behavioural property 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 macro capabilities to help us build the formula:

```
macro GETVALUE_1_MEMORY () =
  "GetValue_JMXIndicatorsMethod_Push_1 !memory_usage (Unstable)"
end_macro

macro PUSH_1_MEMORY ()      =
  ("Push_Event (FirstGateway, UnstableMemoryUsage)")
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 the macros defined, we can now write the 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 the 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 Fig. 12, for each client reconfigurable interface there exists a *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

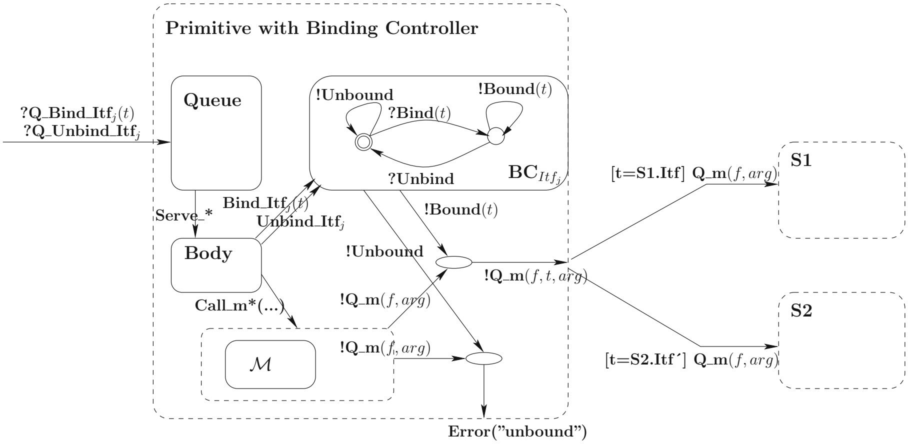

Fig. 12. Binding controller

Table 4. Gateway with reconfigurable interface

	States	Transitions	File size (gb)
hmgateway-reconfig.bcg	354.252.868	4.178.400.886	∼8.45
hmgateway-reconfig-min.bcg	354.104.012	4.176.956.686	∼8.54

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 the HM Gateway model discussed in Subsect. 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 Fig. 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 *connectedness*.

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

All the properties proven in Subsect. 4.1 still hold for this new HM Gateway model, with a natural overhead in *model-checking* them in a much bigger state-space. However, for this new model we are more interested in addressing the reconfiguration capabilities. For instance, provided that the 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
```

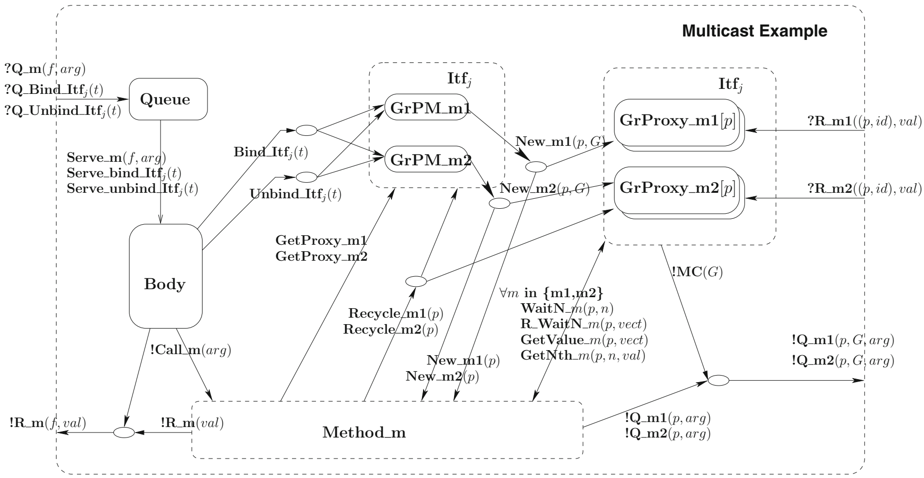

Fig. 13. pNet example for reconfigurable multicast interface

Table 5. Server with reconfigurable *multicast* interface

	States	Transitions	File size (gb)
hmserver-reconfig.bcg	931.640.080	16.435.355.306	∼32.93

The above property is proved *false*. This indicates that provided that the interface is bound, a path yielding an Unbound action without the occurrence of a Q_Unbind_ServerMethod 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 Fig. 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 connectedness status. 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 connectedness.

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

Table 6. Relevant numbers regarding the generated model with reconfigurable interfaces

	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

The generated state-space for the HM Server model nearly attained 1 billion states.[9] 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 state-space, we were still able to *model-check* it against the same properties discussed in Subsect. 4.2.

5.3 Model Generation and Proven Properties

As seen in Subsect. 4.3, building the product of the system already showed to be delicate. Abstraction techniques such as *communication hiding* were already required to build the 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 state-spaces. It mainly acted as a means to leverage the subsequent *minimization* process, allowing for a very significant state-space reduction. Table 6 illustrates the results obtained by following the same approach as above.

We obtained a significant state-space reduction for the HM Gateway model, but we were unable to *minimize* the HM Server. Indeed, *communication hiding* may leverage state-space 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 state-space reduction as well for the HM Server.

While *communication hiding* revealed to be a valuable tool, *minimization* is still a bottleneck if the input state-space 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 ...

[9] As mentioned in Subsect. 4.2, for the HM Server model, the JMX Indicators component is generated with a context not including 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 state-space generation on one of its components can provoke.

Table 7. Relevant numbers regarding the generated model with reconfigurable interfaces

	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	3.054.464.649	38.680.270.695	∼74.16 gb

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

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 connectedness. 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

The maturity attained by the CADP toolbox made it a reference tool among the formal methods community. Several case studies have been published, namely industrial ones addressing other goals than verification. For instance, in [8] Coste et al. discuss performance evaluation for systems and networks on chips. More closely related with our work we must refer the experiments presented in [7]. A dynamic reconfiguration protocol is specified and model-checked, however their focus is on the reconfiguration protocol itself rather than reconfigurable applications.

Indeed, 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 [11] 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 [15]. This work however, focuses on the protocol itself, and not in the behaviour of a reconfigurable application. Moreover, in [10] we presented Mefresa — a ***Me**chanized **F**ramework for **Re**asoning on **S**oftware **A**rchitectures*. This work discusses a formal specification and a (re)configuration language for GCM architectures. All the involved machinery and underlying formal semantics are mechanized in Coq. However, at the current stage of development its main focus is on the 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 a rather unaddressed topic. This can be justified by the inherent complexity that such systems impose. However, reconfiguration plays a significant role for the increase in systems availability, and is a key ingredient in the autonomic computing arena, thus 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. Several lessons can be drawn from this work.

The Spinnaker project gave us the opportunity to promote the use of formal methods within the industry. As expected, the interaction with our industrial partners revealed to be a demanding task. Common budgetary issues (time allocation, hirings, ...) of such projects and lack of prior formal methods' exposure by our partners were some of the barriers to overcome. This was further aggravated by the fact that software development was playing a little part in the overall project budget, and therefore not a main priority.

Nevertheless our experience revealed to be fruitful. We were able to witness the general curiosity on the use of formal methods by the industry, and increase our understanding on the needs and obstacles for its broader adoption. Indeed, collaborative projects of this nature allow the industry to *test the waters* and expose researchers to real-world scenarios. However, bridging the gap between the industry's expectation and the current state of the art still remains as a challenge for the research community. To this end, recent work on Vercors [12] aims at bringing intuitive specification languages and graphical tools for the non-specialists.

Concerning our task at hand, modelling the HyperManager application upto the intricacies of the middleware led us to a combinatorial explosion in the number of states. This, is further exasperated by the inclusion of reconfigurable interfaces. Even the use of compositional and contextual state-space 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. This bottleneck can be alleviated by performing the synchronization product in a *distributed* manner. Alas, this is not supported by the CADP toolbox. Alternatively, CADP supports τ-reduction algorithms that reduce *on-the-fly* the existent τ-transitions. While this approach was successfully applied in [5], its practical effects for this case study remain as future work.[10] Moreover, handling such big state-spaces teaches us the importance of automation regarding model generation. Indeed, debugging can be a daunting task due to the inherent complexity and size of the involved models. Regarding this issue we must refer that we plan on tackling behavioural specification concerns within the Mefresa framework as future work. This will leverage the use of deductive reasoning in a usual model-checking context as demonstrated in [14], and thus relax the burden of dealing with huge state-spaces.

At last, as usual in the realm of formal verification, we conclude that abstraction is the key. Taking advantage of CADP's facilities for *communication hiding*, one can specify actions that need not to be observed for the verification purposes, which further enhances the effects of a subsequent minimization by branching *bisimulation*. This illustrates the pragmatic rationale of formal verification by model-checking — the most likely reason behind its acceptance in the industry.

Acknowledgements. The authors are grateful to Frédéric Lang from the INRIA Vasy team for troubleshooting with the CADP toolbox and the semantics of the Fiacre specification language. Moreover, our *ingénieur experts* Arthur Mbonyinshuti and Bartlomiej Szejna deserve a mention for technical assistance regarding CADP's license management and the implementation of The HyperManager, respectively.

References

1. Ameur-Boulifa, R., Henrio, L., Madelaine, E., Savu, A.: Behavioural semantics for asynchronous components. RR RR-8167, December 2012
2. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)

[10] It is worth mentioning that the immediate concerns and goals for this case study were more aimed at convincing our industrial partners on the ease of use of our verification workflow.

3. Berthomieu, B., Bodeveix, J.P., Filali, M., Garavel, H., Lang, F., Peres, F., Saad, R., Stoecker, J., Vernadat, F.: The syntax and semantics of FIACRE. RR (2009)
4. Bolognesi, T., Brinksma, E.: Introduction to the iso specification language lotos. Comput. Netw. ISDN Syst. **14**(1), 25–59 (1987)
5. Ameur-Boulifa, R., Halalai, R., Henrio, L., Madelaine, E.: Verifying safety of fault-tolerant distributed components. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 278–295. Springer, Heidelberg (2012)
6. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press (2013)
7. Aguilar Cornejo, M., Garavel, M., Mateescu, R., De Palma, N.: Specification and verification of a dynamic reconfiguration protocol for agent-based applications. In: Zielinski, K., Geihs, K., Laurentowski, A. (eds.) New Developments in Distributed Applications and Interoperable Systems. IFIP. Springer, Heidelberg (2001)
8. Coste, N., Hermanns, H., Lantreibecq, E., Serwe, W.: Towards performance prediction of compositional models in industrial GALS designs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 204–218. Springer, Heidelberg (2009)
9. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: a toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
10. Gaspar, N., Henrio, L., Madelaine, E.: Bringing Coq into the world of GCM distributed applications. Int. J. Parallel Program. (HLPP'2013 Special Issue). doi:10.1007/s10766-013-0264-7 (2013)
11. Inverardi, P., Muccini, H., Pelliccione, P.: Charmy: an extensible tool for architectural analysis. In: ESEC-FSE'05, ACM SIGSOFT Symposium on the Foundations of Software Engineering. Research Tool Demos, 5–9 September 2005
12. Kulankhina, O.: A graphical specification environment for GCM component-based applications. Ubinet master internship report, INRIA (2013)
13. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
14. Sprenger, C.: A verified model checker for the modal mu-calculus in coq. In: Steffen, B. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 1384. Springer, Heidelberg (1998)
15. The Coq Development Team. The Coq Proof Assistant Reference Manual (2012)