Observability and Controllability of Wireless Software Components

Fabien Romeo, Franck Barbier, and Jean-Michel Bruel

LIUPPA, Université de Pau et des Pays de l'Adour
Av. de l'Université, B.P. 1155, F-64013 PAU - France
fabien.romeo@univ-pau.fr, barbier@franckbarbier.com, bruel@univ-pau.fr

Abstract. Software components embedded in wireless devices are subject to behavior which cannot be fully and realistically predicted. This calls for a runtime management infrastructure that is able to observe and control the components' states and to make their behaviors explicit, tangible and understandable, in any case and at any time. In this paper, we propose a framework for remotely administrating the functional behavior of software components deployed on wireless nodes. This framework is based on components which are locally managed by internal managers on the wireless side. The controllable nature of components relies on executable UML models that persist at runtime. On the administration side, models are replicated and synchronized with the models that constitute the inner workings of the wireless components.

1 Introduction

Component-based development is a challenging topic in the area of ubiquitous systems. More particularly, this is illustrated by research on specialized component models (*e.g.*, pect [1], koala [2], pecos [3], beanome [4] or frogi [5]) which themselves may support composition techniques that are specific to ubiquitous systems.

Many studies have shown that embedded system developers expect better analysis supports of software behavior. Better testability and debuggability are among these major requirements [6, 7]. Component-based development may be seen as a breakthrough with respect to this topic. Indeed, building software by means of components enables the identification and the setup of deployment properties. As for the compositions of components, they may express links which may reflect wireless infrastructures in a structured and logical way. If one has at one's disposal an appropriate formalism to design the inside of components (implementation) and the outside (interfaces and their dependencies embodying compositions), runtime management may benefit from this formalism. More specifically, this concerns the executable component/composition behavior models that result from using this formalism. Therefore, models act as tracking and monitoring supports.

In the area of ubiquitous systems, mastering deployment conditions includes overcoming some stumbling blocks. Instable communication connections that may be broken, damaged modes are frequent, runtime environments/infrastructures are mobile and may quickly evolve, etc. Thus, emphasizing the management-centric or

model-driven design of software components is not enough. A management system on the top of a distributed application composed of several varied wireless components also requires specific attributes: self-management as defined by autonomic computing [8], special manager roles and distribution of the management layer itself.

In this paper, we describe WMX (Wireless Management eXtensions) [9], an adaptation of JMX, which is the standardized management API and framework in the Java world [10]. Although WMX is the adaptation of JMX for ubiquitous systems, we add in WMX an enhanced support to have "true" manageable software components and compositions. While JMX stresses the management infrastructure (inspired by norms like GDMO - Guidelines for the Definition of Managed Objects), it does not provide a component design method. This means that the inside of these components, at any time, may not really be interpretable and intelligible by management systems; these being human or autonomic. Like JMX, we offer a coercive framework in which components comply with design rules so that they may be deployed in WMX-compliant environments. This point mainly relies on the idea of embedded internal managers which interact with the management side. Components are in particular endowed with dedicated management interfaces in order to sort out what is and has to be managed.

Contrary to JMX, we organize and implement the inside and thus the behavior of components based on executable UML 2 State Machine Diagrams, a variant of Harel's statecharts [11]. To enable the persistence of these models at runtime, we have a J2ME (Java 2 Micro Edition)-compliant library which includes and organizes observation and control activities around the components' abstract states. This includes the dependencies between these states (exclusiveness, orthogonality and nesting) and the logical communications of components (event sending) which embody compositions. Concretely, complex state machines may graphically appear in consoles or GUIs and act as the key entry point for management: forcing states for instance.

To present and explain WMX, this paper first discusses the idea of locally managed components, which are the basis of the proposed infrastructure. Next, the relationships between internal managers and the global management system are described. Finally, a case of composition management is illustrated by means of an example. Before we conclude, synthetic performance measures are listed.

2   Internal Management of Components

We first present the design of a locally managed component, made up of business functionalities embodied in a business subcomponent and a modeled behavior controlled by its internal manager. The correlation between these two subcomponents and the behavior model are detailed in the section 2.2.

2.1   Internal Managers and Business Components

In classical management solutions [8, 13] the application and the management system interact through sensors and actuators – or effectors in the autonomic metaphor. Sensors are used by managers to probe the application and actuators are used to execute application actions.

In CBSE, [14] has defined a specific interface, *the Diagnostic and Management interface*, which provides selective access to the internals of the components for management purposes. Since components communicate through their interfaces, it is natural to specify sensors and actuators as interfaces. Figure 1 depicts, through UML 2 Component Diagrams the resulting architecture of our notion of locally managed component. We have gathered in management ports three types of interfaces acting as sensors and actuators to relay information between the business component and the internal manager inside the locally managed component.
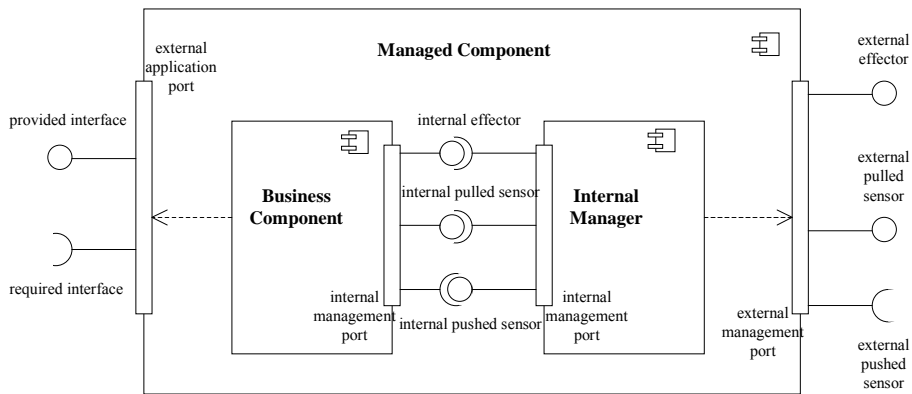

Figure 1. Managed Component Architecture

From a design perspective, we have on one side the business component, which implements the concrete business functionalities, *i.e.* the computation, and on the other side the internal manager, which controls the component according to its defined behavior model. In this way, the internal manager totally encapsulates the control logic, which is then externalized from the business component (as recommended by [15]) to maximize loose coupling between the components. We have thus been able to compose components according to their behavior models [16], but the definition of such a composition mechanism is out of the scope of this paper.

The managed component can also communicate with other external components through classical provided and required interfaces. These interfaces are part of an external application port that is connected to the business component that is responsible for business functionalities. The internal management is connected with an external management port, which is comprised of sensors and actuators, through which the management system can query the manager about its component's states and act on its behavior (see section 3).

2.2   Behavior Model Facilitating the Management of Components

The principle of the management framework is to include a statechart [11] within each managed component's internal manager. This statechart specifies the component's behavior by a set of states and transitions. Figure 2 represents a detailed UML 2 diagram relating to an example of a managed component. Its behavior is defined by the statechart in Figure 3. The detailed component diagram explicits the

interfaces defined in Figure 1 and the implementation classes of this managed component. The behavior of this component is executed by a statechart engine, the *Statechart_monitor* associated with the internal manager.
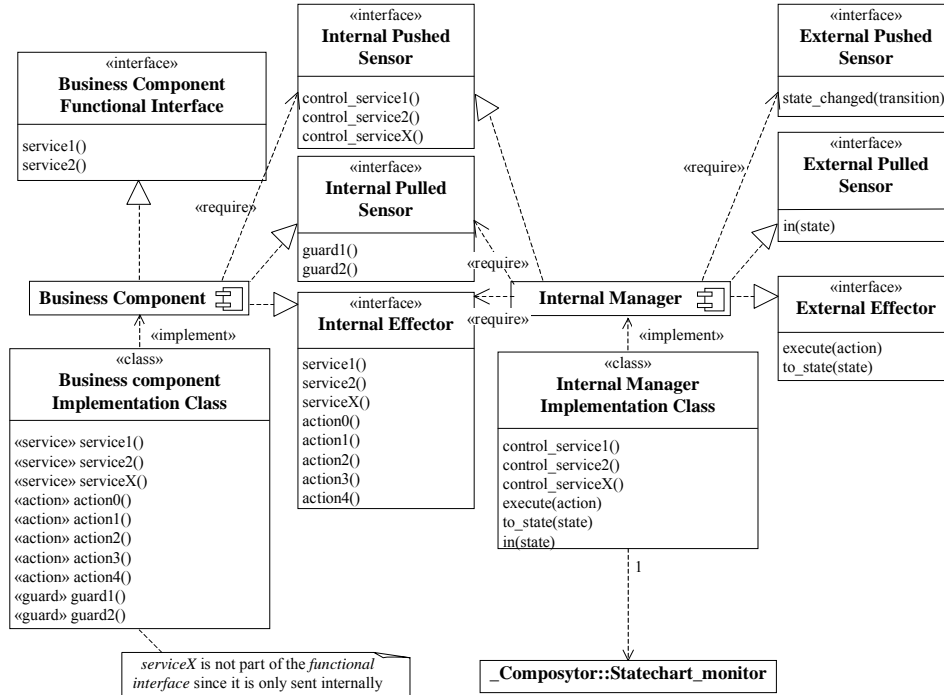

Figure 2. Managed Component's Detailed Architecture

During its execution, this managed component can only be in one of its two mutually exclusive states *SA* or *SB*. According to statechart formalism, *SA* is the initial state. In this state, a request on *service1* exposed in the component's functional interface would generate an event in the internal manager that would trigger a transition from *SA* to *SB*, whereas requests on any other service would have no effect. Conversely, in state *SB* this same event would trigger a transition to *SA*, no matter what substates the component may have. *SB* is a composite state divided into orthogonal regions. At *SB* entry, the component is simultaneously in substates *S10*, *S2* and *S3*, which causes the internal manager to execute in parallel through the internal effector *action0* and *action3* on the business component which implements them. In *S10* substate, a call to *service2* could trigger a transition to *S11* or a transition to *S12* depending on whether *guard1* or *guard2* hold. Note that only one of these two guards can hold simultaneously as specified, if they could hold two at the same time there would have been a consistency error in the statechart due to indeterminism. So if *guard1* holds, *action1* is executed and the component enters into substate *S12*. Notice that it also re-enters into *S2*, as a self-transition is defined for this state upon detection of event *service2*, regardless if *guard1* or *guard2* hold. If *guard2* holds, then a signal is sent to component *self*, *i.e.* to itself, as specified by the following notation ^*self.serviceX*.
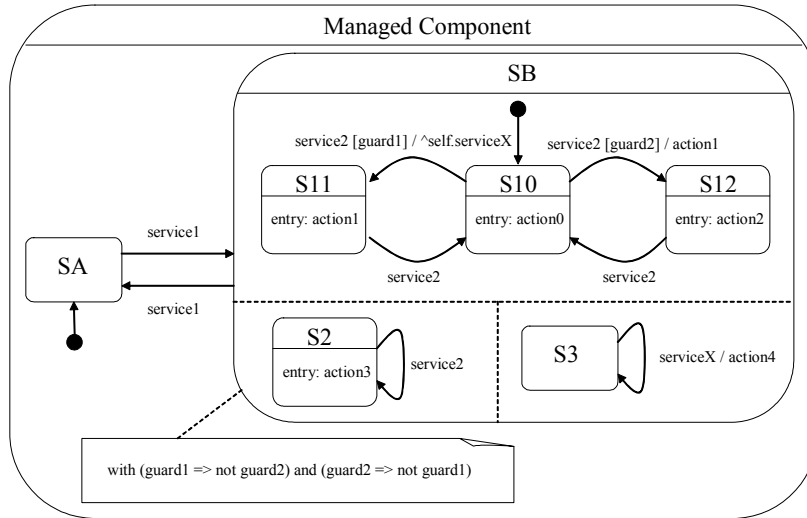
Figure 3. Managed Component Behavior

This example illustrates the relationship between the internal manager and the business component it controls. We can see that two kinds of data need to be captured by the manager: service requests and low-level states. Low-level states are values of objects' attributes that are traditionally monitored in management and are collected here in an abstract way by the evaluation of predefined guards. In management, two different models are used to monitor data: push and pull models [17]. The pull model is based on the request/response paradigm. In this model, the manager sends a data request to the managed host according to its needs, then the managed host replies. Such a sensor, which we call *pulled_sensor*, is used to evaluate the statechart's guards whenever required by adding a provided interface to the business component. Conversely, the push model is based on the publish/subscribe/distribute paradigm. In this model, the manager specifies the data it is interested in, then the managed host is responsible for pushing this data to the manager whenever they change. Thus a *pushed_sensor* is perfectly adapted to collect the business component's incoming events upon reception. We have added a required interface to the business component to equip it with such a sensor.

3   External Management of Components

Management involves two dual activities, monitoring and control. The first part of this section focuses on the way monitoring is considered between a managed component and our management system and the second presents the different control functionalities that are provided.

3.1   Monitoring

Monitoring is the activity of making continuous observations of the evolution of state variables that reflect system dynamics. In the last section, we have seen that the internal manager is responsible for the direct monitoring of the managed component's business activity. But since it is not fully self-manageable, management information needs to be acquired by a higher level management system. In our context of deploying components in embedded systems, the management system has to perform wirelessly, away from managed components. The reason for not integrating this management system into the application system itself is two-fold. First, as we are in a wireless context, we aim at avoiding the overload of wireless devices with heavy management computation. Second, the user interfaces of such systems, often mechanical, are minimal when they exist and thus are not appropriate for management activity.

Hence, we choose to replicate the behavior, *i.e.* the statechart, of managed components on the management side. In managed component internals, the data we managed are events and low-level states (as shown in section 2). A first approach is to reproduce the same scheme. In [18] we forwarded only the events and not the low-level states, which would have been too heavy and inefficient since we do not need to know every change in this data. But this caused synchronization problems since the value of this data is used in guards for firing transitions. As a result, we could not deduce all the transitions that were actually fired.

In order to avoid this problem, we now forward fired transitions instead of events. Hence, we ensure that the replicated statechart evolves in the same way as the original does. In addition, there is no need for the management system to know about low-level states, since the transition choice is already carried out by the internal manager. Data is abstracted to a higher level and the management system only requires the statechart's states in order to work. To allow this communication between the managed component and the management system, we have once again the same two possible models we used in section 2, namely push and pull models. Therefore, we have added an *external_pushed_sensor* as a required interface to the managed component, so that it can notify the management system of any state change. We have also added an *external_pulled_sensor* for re-synchronization purposes in case of communication breakdown. What we have described above is only the information transferred from a running management session. A protocol for starting the process of replication can be worked out, but it is out of the scope of this paper.

3.2   Control

The boundaries of control activity are hard to define because it is involved both in business activity and management activity. Every application has its own control logic and behavior, which coordinates its different functionalities. Control in management interferes with this control logic to activate such or such functionality. In the managed component, we have delegated the whole control responsibility to the internal manager. Contrary to classical applications, in which the control logic is combined with business functionalities, the behavior of our managed component is explicitly de-

fined in a statechart that is directly executed by the *Statechart_monitor* of its internal manager. The latter in turn triggers the corresponding actions on its business component. This allows the internal manager to propose a specific interface to the management system, the *external_effector*, in order to inflect the component's behavior.

Our management system supports three types of control:

- control by event: an event corresponding to a request of service from the component's functional interface is sent to the managed component. This is equivalent to what could be done by a component's client.
- control by state: the managed component is forced into a specified state defined in its statecharts. The control induced by the statechart's transitions is bypassed to put the component directly into the desired current state.
- control by action: it provokes the direct execution of an action in the business component of the managed component without making any change in its current behavior state.

4   Management of Compositions

In the previous two sections, we have seen how management is provided with abstract knowledge of managed components' behavior through their internal managers. This enables high-level management policies for an assembly of managed components, which otherwise could not be taken into account by the internal managers themselves. We first describe a special type of behavior composition used in component based modeling. We then show a management policy for this type of composition that maintains the consistency of the application's overall behavior at runtime.

4.1   Behavior Composition

In CBSE, a software system is considered as an assembly of components. The focus is on practical reuse through the building of new solutions by combining external and home made components. However, building systems from existing parts is known to be a difficult task, especially due to architectural mismatching [19]. In order to represent compound behaviors, Pazzi proposes the adoption of Part-Whole Statecharts (PWS) [20]. In his proposal, compounds' (or parts') behaviors, which are specified by statecharts, are composed through the parallel AND mechanism, which yields a global automaton containing all the compounds' statecharts in different orthogonal regions. An additional region representing the composite's (or the whole's) behavior is added to this automaton. The composite controls its compounds by event sending, but is not notified of its compounds' state change. This could lead to the desynchronization of the composite's statecharts with regards to its compounds' statecharts. Pazzi deals with the problem by obliging the encapsulation of the compoundss. But in [21]'s definition of several forms of composition, the encapsulation property is not a systematic characteristic of this relationship and thus the behavior of the compounds and the composite can diverge. In the following part, we show an example of how a management policy can detect this particular scenario and automatically handle it.

4.2   A Management Policy to Ensure Rigorous Behavior Composition

Let's consider a traffic light component made up of three light components, a red, a yellow and a green one. These components are involved in a relationship where the traffic light is the composite and the lights are the compounds. All the lights have the same behavior, which has two states, *On* and *Off*, as represented by the statechart of Figure 4.
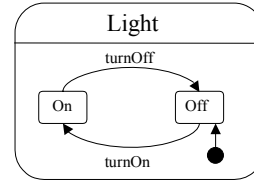
Figure 4. The Light's Behavior

The behavior of the traffic light is depicted by the statechart of Figure 5. It is composed of three main states *Red, Yellow*, and *Green*, and is set to *Red* by means of the *Start* state. When a transition is triggered, it sends signals (notation: *^component.signal*) to switch on or off appropriate lights in order to light only the correct light named by the state that has been reached by the transition.
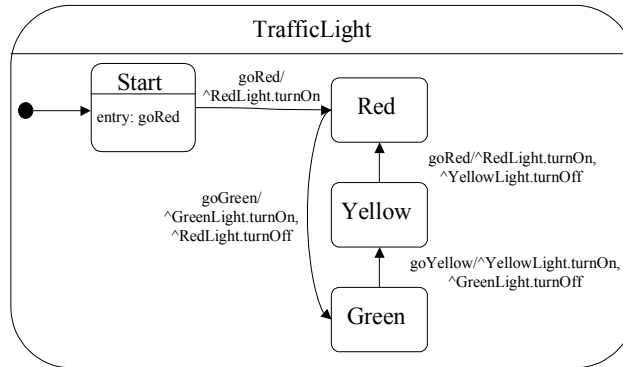
Figure 5. Traffic Light Behavior

Specified like this, the system works well as long as the control of the compounds only comes from the traffic light component, the composite. Indeed, if for any reason, such as an unforeseen event, a hack attack, or a management operation, a light changes its state without the traffic light that initiated it, the behaviors of the composite and its compounds would be desynchronized. This is an illustration of the previously described problem.

To handle this situation, we build, thanks to our framework, these four components as managed components executing the statecharts of Figures 4 and 5. Then we build their corresponding external managers, which replicate the statecharts of the components and allow to control them through the management system. This is depicted with the orthogonal states *Monitor* and *Control* in the managers' behavior specification of Figures 6 and 7.

This allows us to define a management policy in the management system based on the informations provided by these managers. The idea is to specify composite's states as abstract states that belong to a subset of the Cartesian product of the compounds' states. In our example, the traffic light is composed of three lights and the behavior of each light is composed of two states. The Cartesian product yields 2^3 states and only

three are defined for the traffic light, namely *red light on only*, *yellow light on only* and *green light on only*. Other states, in which more than one light are on, are undefined for the traffic light. The next table summarizes this situation.

Components	Valid States		
RedLight	On	Off	Off
YellowLight	Off	On	Off
GreenLight	Off	Off	On
TrafficLight	Red	Yellow	Green

Table 1. States mapping between composite and components

Hence, the composition between the traffic light and its lights can be qualified by two states, *Defined* or *Undefined*, depending on whether the states of the lights reflect a valid state for the traffic light or not (see *valid_state_guard* in Figure 6). The *Undefined* state indicates to the management system that the assembly of components is in a state that has not been designed. It has to be handled manually or autonomically by another management policy, which could reset all the components in a proper state for instance. If the compounds are in a defined state for the composition, the manager of the composite checks if its managed component is synchronized with this state. If not, the manager autonomically sets the composite to the corresponding state (see *consistency_guard* in Figure 6).
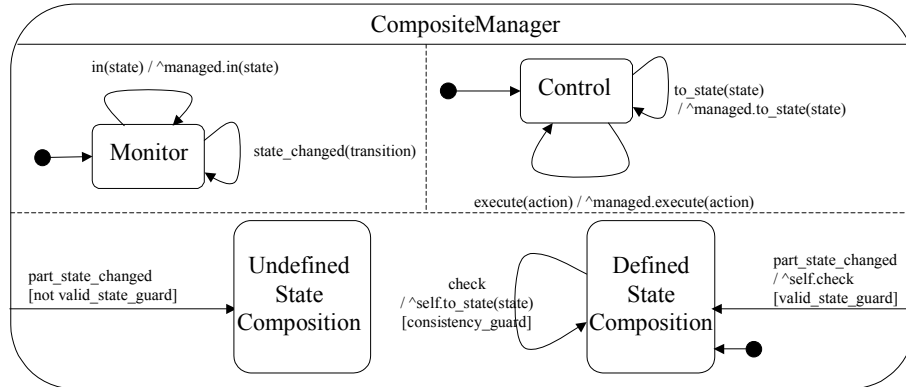
valid_state_guard: (RedLight.in(On) ∧ YellowLight.in(Off) ∧ GreenLight.in(Off))
              ∨ (RedLight.in(Off) ∧ YellowLight.in(On) ∧ GreenLight.in(Off))
              ∨ (RedLight.in(Off) ∧ YellowLight.in(Off) ∧ GreenLight.in(On))

consistency_guard:
(state = Red) ⇒ (managed.in(Red) ⇒ (RedLight.in(On) ∧ YellowLight.in(Off) ∧ GreenLight.in(Off))
∨
(state = Yellow) ⇒ (managed.in(Yellow) ⇒ (RedLight.in(Off) ∧ YellowLight.in(On) ∧ GreenLight.in(Off))
∨
(state = Green) ⇒ (managed.in(Green) ⇒ (RedLight.in(Off) ∧ YellowLight.in(Off) ∧ GreenLight.in(On))

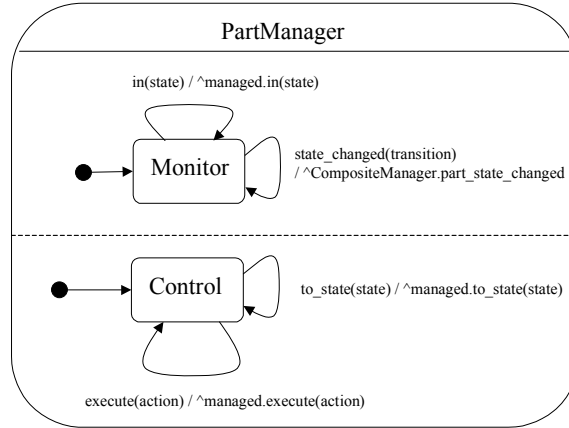**Figure 6.** Composite Manager's Behavior

Figure 7. Compound Manager's Behavior

5   Implementation

The implementation of the presented infrastructure is named WMX, which stands for Wireless Management Extensions. It has to be seen in as an overall effort to rigorously develop component-based complex systems. WMX is part of a framework dedicated to the development of autonomic component-based applications. It is based on a Java library that enables the execution of Harel's Statecharts: the PauWare library [16]. In WMX, both internal and external managers are built on top of this library: internal managers use the J2ME version, called Velcro, and external managers use the J2SE standard version. Communications between these components have been generalized and they are delegated to specific adapters, which support the chosen wireless technologies (Wifi, Bluetooth, WMA, ...). The overall management system relies on the management standard JMX and thus can be incorporated into existing JMX-compliant management solutions.

5.1   Wireless Software Components

WMX provides the necessary facilities to directly implement managed components as specified in Figure 1. From a design viewpoint, this simply leads to extending the *WMX_component* class provided by WMX and to incorporating the statecharts controling its behavior by using the Velcro library. Here is the code of the Light component in Figure 4 (the code is incomplete):

```
public class Light extends WMX_component {
    protected AbstractStatechart _On;
    protected AbstractStatechart _Off;
    protected AbstractStatechart_monitor _Light;
    public Light() throws Statechart_exception {
        // init states
        _On = new VelcroStatechart("On");
        _Off = new VelcroStatechart("Off");
```

```
        _Off.inputState();
        _Light = new VelcroStatechart_monitor(
                        _On.xor(_Off),"Light");
          registerStatechart_monitor(_Light);
        // init transitions
        _Light.fires("turnOn",_Off,_On,[...]);
        _Light.fires("turnOff",_On,_Off,[...]);
    }
    [...]
}
```

In the above code, *Light* is composed of *On* and *Off* states using the XOR operator and it is declared as a statechart monitor, which is the access point to the overall statechart of the Light component. The *registerStatechart_monitor* method (in bold print), which is a member of *WMX_component* class, effectively registers the statechart monitor to be used for management purposes. Then all the management communication matters are automatically handled by the *WMX_component*.

Events in the statecharts are implemented as method calls which notify the statechart monitor to start a run-to-completion process to execute eligible transitions:

```
public void turnOn() throws Statechart_exception {
    _Light.run_to_completion("turnOn");
}
public void turnOff() throws Statechart_exception {
    _Light.run_to_completion("turnOff");
}
```

When declaring a transition between states with the *fires* method, it is possible to specify a guard that will have to be satisfied in order to trigger the transition and an action to be performed when the transition is actually triggered. Here is the signature of the *fires* method:

```
public void fires(java.lang.String event,
                  AbstractStatechart from,
                  AbstractStatechart to,
                  boolean guard,
                  java.lang.Object object,
                  java.lang.String action,
                  java.lang.Object[] args)
        throws Statechart_transition_based_exception
```

In the above signature, it is important to notice that the object in charge of the execution of the action can be specified. In this way, components deployed in the same JVM and can communicate asynchronously through their statechart monitors.

5.2   Wireless Management Communication and Remote Management System

In our proposition, the statechart of a managed component deployed on a wireless device is replicated and kept up to date in its remote management system. The replicated statechart is also implemented by using the PauWare library, but only the states of the original statechart are duplicated; not the transitions. The triggered transitions are directly forwarded by the managed component and there is no event processing to execute the eligible transitions.

In WMX, management communication is done through *Wireless Communicators* which target specific wireless networks such as WiFi, Bluetooth, or WMA (SMS) for

instance. Like this, depending on the available network, one can choose to connect such or such communicator to one's managed component and corresponding manager. Of course our framework depends on the reliability of the wireless network that is used. However in our current implementation, even if communications are temporarily broken, the management system will eventually be updated since our statecharts support asynchronous communications. Moreover, we have deployed the TrafficLight case study on a PDA, which is an HP iPAQ hx4700 embedding J9 Java virtual machine from IBM, using Wifi and the application goes perfectly well, as long as the device remains within the network range. And if it loses connection for a moment the management system restarts in the current state of the managed component.

Lastly, managers in WMX are implemented as MBean in order to be accessible through JMX, which is the standard for management in the Java Platform. Thus, WMX components are manageable through common management systems such as the JMX console or even through a simple web page by using the JDMK HTML adaptor.

6   Performance Issues

In order to evaluate our framework, we employ a benchmark to quantify the execution time overhead per state change. For our purpose, iterations of 100000 state changes are performed on different test components. Table 2 reports the results from this experiment on our test system: a Pentium M 1,6GHz processor with 512 Mo of RAM running Java 1.5 on Windows XP. We chose this system over a handheld device in order to compare WMX with JMX, which can not be run on Java ME. Moreover, this choice also allows us to quantify the cost of the adaptation of PauWare for wireless systems in Velcro.

Implementation	Benchmark	Overhead per state change
Pure Java	2 ms	0 μs
Java + reflect API	14 ms	0,12 μs
JMX (internal access)	721ms	7,19 μs
PauWare (w/o cache)	1491 ms	14,89 μs
PauWare (w cache)	1027 ms	10,25 μs
Velcro (w/o cache)	1529 ms	15,27 μs
Velcro (w cache)	1038 ms	10,36 μs
Following implementations include I/O or networking		
Pure Java + System.out.print()	2584 ms	25,82 μs
WMX (velcro + sockets)	3893 ms	38,91 μs
JMX + RMI connector	22077ms	220,75 μs

Table 2. Benchmarks

At first glance the results show that PauWare is twice heavier than JMX, but this is acceptable when considering that the State Machine engine performs a lot more

controls than JMX. Moreover, the performances of Pauware are improved by the use of cached transitions: the transitions that are not dynamically resolved at runtime can be statically defined once and for all. Another interesting result is that the adaptations made in Velcro to render the State Machine engine compliant with Java ME do not much affect the performance.

At last in more realistic situations, *i.e.* when the management involves logging or networking, WMX is only 50 percent slower than a simple log console (Pure Java + System.out.print()) and it clearly outperforms JMX used with an RMI connector.

7   Conclusion

In this paper, we have presented a management system for software components deployed in wireless embedded systems. The solution focuses on the management of model-driven behaviors. To that end, we have introduced internal managers which are responsible for observing and controlling managed component behaviors. Thanks to these wireless-side managers, we have shown how the global management system is organized. More precisely, we have illustrated the exchanges flows induced by management activities. Then, we have described an example of management policy based on a particular type of composition. Finally, performances issues were briefly evoked.

At this time, we have experimented and validated our approach by a prototype running on real devices (PDAs especially). The wireless management side is obviously based on J2ME and PauWare (the support for executable UML 2 State Machine Diagrams). As for the global implementation of the prototype, we have kept JMX on the non-wireless side in order to take advantage of all of the features of this standard. Our existing implementation is not bound to any specific running environment or component model. We on purpose are currently investigating the OSGi platform which has become highly used in wireless systems.

We are also currently working on "autonomous" management policies that might rely on our system to make management activities more and more autonomic. Clearly, self-healing for instance, a kind of fault recovery mechanism, might take advantage of rolling back state machines to stable consistent configurations when abnormal situations exist or persist. Self-configuration may also be more easily and more straightforwardly instrumented by forcing states of components.

References

1. Wallnau, K.C.: Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (2003)
2. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. Computer 33(3) (2000) 78–85
3. Winter, M., Genssler, T., Christoph, A., Nierstrasz, O., Ducasse, S., Wuyts, R., Arvalo, G., Mller, P., Stich, C., Schnhage, B.: Components for Embedded Software

– The PECOS Approach. In: Second International Workshop on Composition Languages, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP), Malaga, Spain (2002)

4. Cervantes, H., Hall, R.S.: Beanome: A Component Model for the OSGi Framework. In: proceedings of the Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices, Lausanne, Switzerland (2000)

5. Desertot, M., Cervantes, H., Donsez, D.: FROGi: Fractal components deployment over OSGi. In: 5th International Symposium on Software Composition SC'06, Vienna, Austria (2006)

6. Crnkovic, I.: Component-based Software Engineering for Embedded Systems. In: International Conference on Software engineering, St. Luis, USA, ACM (2005)

7. Möller, A., Fröberg, J., Nolin, M.: Industrial Requirements on Component Technologies for Embedded Systems. In: International Symposium on Component-Based Software Engineering, Edinburgh, Scotland, Springer Verlag (2004)

8. Kephart, J., Chess, D.: The Vision of Autonomic Computing. In: Computer Magazine. Volume 36. IEEE Computer Society (2003) 41–50

9. Romeo, F.: WMX, http://www.univ-pau.fr/~fromeo/wmx

10. Kreger, H., Harold, W., Williamson, L.: Java and JMX, Addison Wesley (2003)

11. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8(3) (1987) 231–274

12. Grieskamp, W., Heisel, M., Dörr, H.: Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. Lecture Notes in Computer Science 1382 (1998) 88–115

13. Buzato, L.E.: Management of Object-Oriented Action-Based Distributed Programs. PhD thesis, University of Newcastle upon Tyne (1994)

14. Kopetz, H., Suri, N.: Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In: 6th IEEE International Symposium on Object-oriented Real-Time Distributed Computing, Hokkaido, Japan (2003)

15. Lau, K.K., Elizondo, P.V., Wang, Z.: Exogenous Connectors for Software Components. In: Eighth International SIGSOFT Symposium on Component-based Software Engineering, Springer Verlag (2005)

16. Romeo, F., Ballagny, C., Barbier, F.: PauWare : un modèle de composant basé état. In: Journées Composants, Canet en Roussillon, France (2006) 1–10

17. Martin-Flatin, J.P.: Push vs. Pull in Web-Based Network Management. In: Proc. 6th IFIP/IEEE Intl. Symposium on Integrated Network Management (IM'99), Boston, MA (1999) 3–18

18. Romeo, F., Barbier, F.: Management of Wireless Software Components. In: the 10th International Workshop on Component-Oriented Programming in the 19th European Conference on Object-Oriented Programming, Glasgow, Scotland (2005)

19. Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch or Why it's hard to build systems out of existing parts. In: 17th International Conference on Software Enginneering, Seattle, Washington, ACM SIGSOFT (1995) 179–185

20. Pazzi, L.: Part-Whole Statecharts for the Explicit Representation of Compound Behaviors. In: UML. (2000) 541–555

21. Barbier, F., Henderson-Sellers, B., Parc, A.L., Bruel, J.M.: Formalization of the Whole-Part Relationship in the Unied Modeling Language. IEEE Trans. Software Eng. 29(5) (2003) 459–470