Abstract—Software components embedded in ubiquitous systems, especially those using wireless networking, are subject to unpredictable behaviors inherent to using these systems in the physical world. This calls for a runtime management infrastructure to observe and control the components’ states and consequently their resulting behaviors. In this paper, we propose a framework for remotely administrating the functional behaviors of software components deployed on wireless devices. This framework is based on components locally managed by internal managers. At the crossroads of the administration system and the wireless application itself, these internal managers master the components’ actions by interpreting executable models which themself implement the components’ expected behaviors. Besides, high-level managers define and apply management policies to the application’s global behavior through the use of these internal managers. Such a coordination occurs through composition.

I. INTRODUCTION

Component-based development is a hot topic in the embedded system domain, as illustrated by the numerous research investigations for designing software of such systems as assemblies of components (e.g., pect [1], koala [2], pecos [3], beanome [4], or frogi [5]).

Many studies have shown that embedded system developers expect a better analysis of software behavior: better testability and debuggability of components are among the major requirements [6], [7]. These matters are reemphasized by the recent ubiquitous shift of embedded systems, which are more than ever subject to unexpected behavior due to their physical integration into an evolving environment [8]. So even if software components’ behavior is intensively tested at development time, ubiquity generates enhanced management requirements at runtime. This is where autonomic computing comes into play [9].

Autonomic computing is a new approach which aims at building computing systems that can self-manage. Its ultimate goal, which seems utopian today, is to totally remove human intervention from the maintenance process by providing software with the ability to self-configure, self-optimize, self-heal and self-protect. Some frameworks, such as [10], enable the development of autonomic component-based applications by defining management rules based on components’ internal variables. These rules, which are conditional expressions, are interpreted at runtime by management agents in order to control the application’s behavior. Although they allow to automate some activities in the management process, there is a major drawback in using these frameworks. In [10], there is no design technique and support (dedicated modeling constructs, ways of deriving these constructs into concrete management code, etc) that allow us to instrument their autonomic management philosophy. In the domain of embedded systems, the need for such design technique and support is required.

In order to facilitate the design of autonomic component-based embedded software, enhancing the management of such systems, we propose an infrastructure in which components are controlled by internal managers through executable models of the components’ functional behavior. For this we use UML 2 State Machine Diagrams, a variant of Harel’s statecharts [11], which is a recognized modeling construct in the domain of embedded systems [12]. Because these models are executable, the abstraction effort realized at development time leads to concrete software artefacts accessible at runtime.

This paper presents first the locally managed components, which are the base of our infrastructure, next it presents the relationships between the internal managers of these components and the global management system. Finally, a case of composition management is illustrated in an example; before we conclude and expose perspectives about this work.

II. INTERNAL MANAGEMENT OF COMPONENTS

We first present the design of 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.

A. Internal Managers and Business Components

In classical management solutions [9], [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.

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 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 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).

**B. 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.

During its execution, this managed component only can 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*, 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’.

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.

**III. 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 which are provided.

**A. 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 aims 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 low-level states, which would have been too heavy and inefficient since we do not need to know every change of these data. But this caused synchronization problems since the value of these 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 needs the statechart’s states in order to work. To allow this communication between the managed component and the management system, we have 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 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 loss. 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.

B. Control

The control activity is hard to delimit 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 defined 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_efector, 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 are bypassed to put the component directly into the desired current state. This is equivalent to having a transitive closure on the flat state graph which corresponds to the statechart of the component.
- 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.

IV. MANAGEMENT OF COMPOSITIONS

In the last 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 that keeps at runtime the consistency of the application’s overall behavior according to this composition.
Fig. 2. Managed Component Detailed Architecture
A. 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 extra region representing the composite’s (or 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 imposing 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.

B. 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.

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: \( \text{component.signal} \)) 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.

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.

<table>
<thead>
<tr>
<th>Components</th>
<th>Valid States</th>
</tr>
</thead>
<tbody>
<tr>
<td>RedLight</td>
<td>On Off Off</td>
</tr>
<tr>
<td>YellowLight</td>
<td>Off On Off</td>
</tr>
<tr>
<td>GreenLight</td>
<td>Off Off On</td>
</tr>
<tr>
<td>TrafficLight</td>
<td>Red Yellow Green</td>
</tr>
</tbody>
</table>

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 the management system that the assembly of components is in a state that has not been designed. It has to be handle manually or autonamically 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 autonamically sets the composite in the corresponding state (see consistency_guard in Figure 6).

V. IMPLEMENTATION

The implementation of the presented infrastructure is named WMX\(^1\), which stands for Wireless Management Extensions. It has to be seen in an overall effort for rigorously developing 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\(^2\) [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 supporting 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.

A. 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 extend WMX_component class provided by WMX and to incorporate the statecharts controlling its behavior using Velcro library. Here is the code of the Light component in Figure 4 (code is incomplete):

```java
public class Light extends WMX_component {
    protected AbstractStatechart _On;
    protected AbstractStatechart _Off;
    protected AbstractStatechart_monitor _Light;

    public Light() throws Statechart_exception {
        _On = new VelcroStatechart("On");
        _Off = new VelcroStatechart("Off");
        _Off.inputState();

        _Light = new VelcroStatechart_monitor(
            _On.xor(_Off),"Light");

        registerStatechart_monitor(_Light);

        // init states
        _On = new VelcroStatechart("On");
        _Off = new VelcroStatechart("Off");
        _Off.inputState();

        // registerStatechart_monitor(_Light);

        // init transitions
        _Light.fires("turnOn",_Off,_On,[...]);
        _Light.fires("turnOff",_On,_Off,[...]);
    }
}
```

\(^1\)available at http://www.univ-pau.fr/~fromeo/wmx

\(^2\)available at http://www.pauware.com
valid state guard: (RedLight.in(On) \land YellowLight.in(Off) \land GreenLight.in(Off)) \\
\lor (RedLight.in(Off) \land YellowLight.in(On) \land GreenLight.in(Off)) \\
\lor (RedLight.in(Off) \land YellowLight.in(Off) \land GreenLight.in(On))

consistency guard: 
- (state = Red) \Rightarrow (managed \land \text{RedLight.in(On) \land YellowLight.in(Off) \land GreenLight.in(Off)}) \\
- (state = Yellow) \Rightarrow (managed \land \text{YellowLight.in(On) \land GreenLight.in(Off) \land \text{RedLight.in(Off)}}) \\
- (state = Green) \Rightarrow (managed \land \text{GreenLight.in(On) \land \text{RedLight.in(Off) \land YellowLight.in(Off)}})

Fig. 6. Composite Manager’s Behavior
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 purpose. Then all the management communication matters are automatically handle 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:

```java
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 must be satisfied to trigger the transition and an action to perform when the transition is actually triggered. Here is the complete signature of the fires method:

```java
public void fires(java.lang.String event,
        AbstractStatechart from,
        AbstractStatechart to,
        boolean guard,
        java.lang.String action,
        java.lang.Object object,
        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.

B. 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 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, wireless 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 used. However in our current implementation, even if communications are temporary 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 in the network range. And if it lose 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.

VI. Conclusion

In this paper, we have presented a management system of software components deployed in wireless embedded systems. Our solution focuses on the management of the functional behavior of the components. To that end, we have designed internal managers responsible for controlling the behavior of managed components by means of executable statecharts.

Thanks to these abstract models of their behavior, we have seen how the remote management system can efficiently monitor and control them. Lastly, we have described an example of management policy based on high-level consideration of the behavior of the components which are involved in a particular type of composition.

We have validated our approach by a prototype running on real devices and implemented for the Java Platforms, i.e. the mobile edition for the managed side and the standard edition with JMX for the management side. This implementation is not bound to any specific component model, but we plan to target it for the OSGi platform, which has become highly used in wireless systems.

We are also currently working on other management policies that could be based on our system in order to make management activity more and more automated. Moreover, coupling our system with other autonomous systems would be interesting also.

Another interesting topic is in the separate design of components from a business perspective and from a behavioral perspective. Here, we have separated these two facets into two different sub-components of our managed component. The aspect paradigm seems to be an elegant and appropriate solution to compose these two parts and it would merit further investigation.

REFERENCES


[16] F. Romeo, C. Ballagny, and F. Barbier, “PauWare : un modèle de composant base etat / PauWare: a State-Based Component Model,” in Journées Composants / Components Days, Canet en Roussillon, France, October 2006, pp. 1–10.


