Handbook of Research in Mobile Business: Technical, Methodological, and Social Perspectives

Bhuvan Unhelkar
University of Western Sydney, Australia

Idea Group
REFERENCE

IDEA GROUP REFERENCE
Hershey · London · Melbourne · Singapore

Chapter XV
Administration of Wireless Software Components

Franck Barbier
PauWare Research Group, France

Fabien Romeo
PauWare Research Group, France

ABSTRACT

Software components embedded in mobile and wireless devices, as ordinary components deployed in distributed systems, have to be managed in order to recover faults, to trace and analyze behaviors, to enable business services such as online maintenance, customer practice understanding and so on. Despite the existence of management standards and platforms, the implementation of management facilities inside components as well as the possibility to access and operate these facilities by means of appropriate interfaces (a configuration interface for instance in order to instrument dynamical re-configuration) are not actually available. In this scope, this chapter discusses and provides a design method and an associated Java library in order to have manageable and self-manageable components specific to mobile and wireless environments.

INTRODUCTION

A major trend concerning the mobile system industry is the need for designing software applications as assemblies of reusable components, such as Java components in J2ME environments and C# components in Windows CE environments. Components are interconnected through well-defined interfaces, while their implementation remains hidden. This results in increased reuse, easy isolation of faults, and overall improvement of quality and reliability. Furthermore, it also enables components to be independently deployed by third parties. Deployment occurs on various devices such as mobile phones, personal digital assistants

(PDAs), set-top boxes, smart cards, and so on. Owing to the fact that mobile deployment environments are different from development environments, abnormal behaviors and/or misuses occur. Consequently, some deferred assessment is only possible if components have been endowed with remote administration capabilities, including state and behavior supervision, even control, at the time components face unstable execution contexts.

From a business perspective, such a global approach is a way for supporting and thus for offering better customer services: better start-up processes, analysis of deficiencies, possible corrections through (re)configurations, marketing-based studies of common practices and expectations, and so forth. All of the well-known qualities of the component-based development technology (National Coordination Office for Information Technology Research and Development, 2001; Szyperski, Gruntz, & Murer, 2002) seem promising to achieve the necessary flexibility and adaptability imposed by the mobile systems' market.

The purpose of this chapter is to propose appropriate concepts, techniques, and tools to manage component behaviors and their associated interactions within wireless devices. Management activities stress component behavior tracing and possible dynamic (re)configuration in order to ensure actual supervision and control of mobile devices. Some of the contribution comes from the results of the Component+ European project (www.component-plus.org) in which the idea of Built-In Test (Barbier, 2005) has been formalized, developed, and put into practice for multimedia software components.

This chapter extends some of the ideas relating to the BIT technology that initially does not support administration facilities. We especially deal with the notion of "wireless software component." Even if this expression may *a priori* not make sense, we here mean software components deployed in wireless systems. The chapter is organized in three main parts. The first part reviews the current relation between CBSE and software for mobile systems. After justifying an interesting potential synergy between the two domains, a discussion about component management architecture standards as Java Management eXtensions or JMX (Sun Microsystems, 2002; Kreger, Harold, & Williamson, 2003) is proposed. This first part ends with a focus on UML 2 in particular and executable modeling language in general, in order to answer the question: What could be a suitable technical framework for wireless component administration?

While the second part of the chapter exposes our contribution in terms of executability of the UML 2 notation and its inherent implementation for component administration, the third and final part deals with a fully implemented case study—a home automation system, including a complex wireless software component corresponding to a programmable thermostat. The state machine diagram of this component is in the Appendix. For illustration purposes, screen shots are offered, especially those relating to the programmable thermostat's management activities in Web browsers.

SOFTWARE FOR MOBILE AND WIRELESS DEVICES: AN OVERVIEW

Nowadays, a great diversity of mobile devices is offered to consumers in different market segments such as telephony, digital interactive television, home automation, and automotive industry. Each device category uses specific hardware architectures and equipment to better fit customer requirements. For instance, PDAs use more powerful processors than mobile phones and also provide larger screens; and car navigation systems, for instance, use voice recognition, while cheaper devices often support simpler software functionality. Despite the existence of varied functionality, software for wireless devices primarily benefits from being constructed on the top of recognized and stable platforms such as J2ME in the Java world. Next, the existence of norms permits laying down assumptions for programmers so that their code does not deal with a large number of technical development/deployment contexts. For user interfaces of PDAs and cell phones, for instance, the MIDP (mobile information device profile) norm (Bloch & Wagner, 2003) is in this scope a relevant framework.

However, new software paradigms such as autonomous computing may make software components not aware of their computing environment (e.g., new possible interactions with unknown components, expected dynamic adaptations). Since this problem cannot be tackled at development time, but only when the system is deployed, there is a need for extra code that may anticipate such potential challenges. In this line of reasoning, we have defined the BIT technology that refers to testing code that remains at runtime. As for administration, it refers to the possibility for ruling such embedded code remotely, because of the need for administration policies involving not only a single component but a system of components. In the case of mobile devices, overheads caused by this extra code raise performance problems and thus create the necessity of an adequate management architecture presented in this chapter.

WIRELESS SOFTWARE COMPONENTS

The component technology (Szyperski et al., 2002) has been proven mature for numerous application domains such as Web-based and business applications, and real-time systems. However, its usage has been evolving slowly in the field of mobile systems.

Components indeed run in large infrastructures involving different server types, more or less normalized protocols, and sophisticated services (e.g., timer services, transaction management services), all of these being often incompatible with hardware features of mobile systems that offer a reduced set of possibilities for software deployment. Software components may thus exist within wireless devices, but many interesting system-based services have to be externalized. One may for instance imagine transaction management facilities that can only be implemented on a server-side base with appropriate communication. Applying rollback actions for a wireless software component then becomes not so easily compared to an Enterprise JavaBeans in an ordinary J2EE server. This especially results from the need for transaction management-oriented code that may clutter, even damage, any user-oriented functioning in a wireless device. However, in the field of distributed applications, the need for management of not only component units, but component assemblies, makes software in mobile systems a piece in a puzzle but not a standalone running machine—a wireless device is no more than a node in a distributed application. At least, virtual links have thus to be created with components that run on other nodes (called remote components later in this chapter), but have close collaboration with those on mobile systems.

Wireless software components are in essence reactive systems in the sense that their behavior mostly consists of processing a lot of events of different types (incoming communication flow, user interface interaction, etc.). The high frequency of events and the large spectrum of interpretation contexts show that state machines are good candidates for behavior specification. More generally, building software for mobiles systems with design languages that have interesting features such as (i) being standards, (ii) supporting executability (Mellor & Balcer, 2002), and (iii) supporting compositionality (Bock, 2004), permits one to weave—more clearly and more rigorously—wireless components with server-side components with which they have to collaborate.

ADMINISTRATION

The idea of administration (the term management has been used interchangeably) stems from the network domain in which network element attributes and behaviors have to be supervised and, in case of failure, they also have to be driven so that communication occurs as well as possible: availability of damaged modes, correction actions as switching communication flows, and so forth.

The need for standardization has generated dedicated administration supports, namely dedicated protocols including SNMP (Simple Network Management Protocol) or CMIP (Common Management Information Protocol) and, in the world of Java and software components, the Java Management eXtensions framework.

Administration covers two main activities: supervision (e.g., *in-situ* testing) and monitoring (e.g., dynamical (re)configuration). In the Java global context, network elements that are "resources" that are viewed within application top layers as software components are often named managed objects.

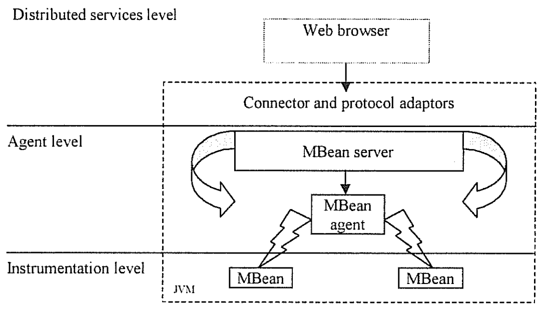
Java Management eXtensions

JMX has been specified and implemented to have an open standard for managing resources based on an abstract approach. JMX high level of abstraction does not rely on specific protocols or other platform-dependent properties. Moreover, the wireless nature of resources and software components may be hidden without changing administration strategies and policies. We here see a concrete advantage of incorporating software components into wireless devices: an uniformed access to resources that is independent of running contexts, as for instance Bluetooth-based protocols and, more generally, any coercive communication mechanism imposed by wireless systems.

JMX architecture is grounded on three main categories of software components (Figure 1):

- MBeans (Managed Beans) are the components that can be managed by JMX. They define standard interfaces in which a high or low number of functional operations (normal behavior) as well as in-house management operations (e.g., state observation and tracing) will be accessible at an administration level.
- MBean agents that are directly deployed in JMX servers are responsible for controlling their locally registered MBeans. An MBeans agent generally copes with a set of MBeans in relation to predefined JMX services to ease management of MBeans. These services are common services such as timer services, but also more relevant ones such as relation services that are peculiar to compositionality management of component systems. We may also notice that JMX owns a metalevel, in the sense that MBean agents are themselves manageable entities at the Distributed services level (see Figure 1).
- Connectors are plugged into MBean agents to make them remotely available to management applications. The communication protocol is specific to a connector type that in essence aims at linking JMX to

Figure 1. JMX stereotyped architecture (Sun Microsystems, 2002)



third-party applications. Figure 1 shows the case of an HTML adaptor that connected some given agents with a Web browser.

Administration of Wireless Environments

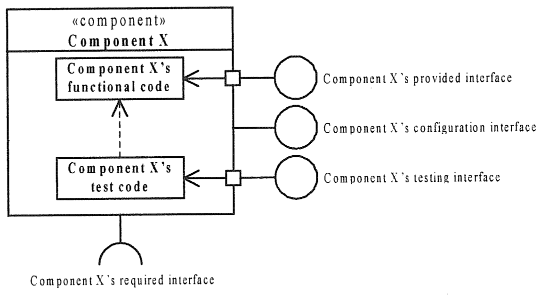
The management of pervasive computing environments is forced by the lack of processing power of smart devices such as cellular phones. In other words, such devices rely on layered distributed systems that aim to coordinate and effectively ensure most of the expected functionality. So, even though the idea of administrating wireless environments is not new, there is no standard and rigorous method for designing wireless software components so that they are really manageable.

An illustration of current problems is, for instance, dynamic (re)configuration. One may imagine such a management-based use case for a mobile device in the context of a business-oriented service, which is here online maintenance. For devices equipped with the very latest technologies, including hardware and software, statistics show than return rates for reparation are close to 20%! This is exclusively due to market pressure that prompts to sell products that are not tested enough and may even be non-mature.

Technically, the delivering of patches resulting from a mandatory, prior, and remote in-depth testing, itself based on resuming and/or on simulating special running contexts, cannot be yet easily instrumented. What indeed does "resetting an embedded component to have an initial (safe) context" or "setting up a component to an accurate context (a well-defined state)" mean?

The novelty of our approach is to design components, their internal parts especially, so that the desired administration operations may be logically described. More precisely, this includes the possibility to formally express what is a stable running context, namely forcing a given wireless component to immediately be in a set of parallel states. This also amounts to reaching a context in safely terminating any data processing, namely monitoring entry and exit operations for states. Our solution is grounded on state machine modeling languages, as well as formal execution (simulation) rules of state machines that are plugged into components. In addition, the *PauWare.Statecharts* Java library is offered to achieve all of these objectives.

Figure 2. BIT style for a wireless software component


«component»
Component X
Component X's functional code
Component X's test code
Component X's provided interface
Component X's configuration interface
Component X's testing interface
Component X's required interface

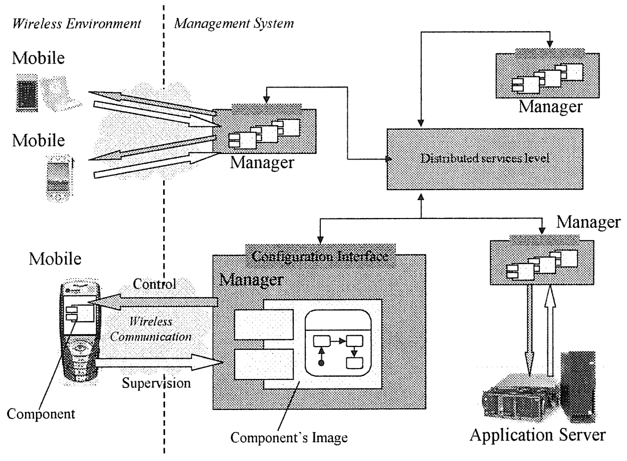
Most of the contribution presented here is an extension of the BIT technology that initially does not address management issues. Briefly speaking, this technology advocates persistent test code in components (see Figure 2) in order to support deployment-based testing. State machines are used for instrumenting model-based testing, and above all, for having the possibility of rapidly prototyping components for assessment purposes. Another major point of BIT is to be a gate for accessing a more or less important quantity of a component's encapsulated part. For management purposes, BIT-based wireless components have required interfaces, functional (a.k.a., provided) interfaces, testing interfaces, and also configuration interfaces that are specific to management (see Figure 2). The two last sorts of interfaces are made up of services whose implementation explores and, in the spirit of administration, possibly changes a component's inside.

UML 2-BASED DESIGN OF WIRELESS SOFTWARE COMPONENTS

An outstanding feature of wireless components is that they have to react to many distinct types of events whose interpretation varies according to several discrete contexts. Regarding graphic user interfaces of smart devices for instance, Horrocks (1999) advocates the use of statecharts (Harel, 1987) for designing and implementing such interfaces. On a cellular phone for example, each physical button is associated with a typed event and a precise context that leads to a specific processing at the time a given function is running under the auspices of the phone's owner.

More generally, in the scope of CBSE, the recent release (version 2) of UML (OMG, 2003a, 2003b) offers all of the necessary and sufficient integrated modeling constructs, including state machine diagrams (i.e., Harel's statecharts), sequence diagrams (useful for communication modeling), and component diagrams for the design of manageable wireless components. All of this happens in the context of the MDA/MDE initiative in which executability promotes lightweight model checking (Mellor & Balcer, 2002) and seamless implementation through model transformation: from platform-independent models as the statechart of a programmable thermostat wireless component in the Appendix to platform-dependent models that, for instance, may refer to specific properties as coercive communication as mentioned above.

Figure 3. Management architecture



A main point of UML 2 state machine diagrams is that they favor any reasoning with *abstract* states that aim at capturing the key situations and the critical phases in which mobile devices require supervision and control. Once again, in the absence of an underlying formalism, management activities cannot rely on precise representations as those offered by state machine diagrams.

TECHNICAL FRAMEWORK

Since supervision and control are dual activities involved in management, one may wonder how both activities have to use component states. In practice, while control implies that managers change—under well-defined conditions and circumstances (see *reset* Java method below)—states of managed components, supervision consists for managers of acquiring information on current states of managed components and clusters of components. Considering that these activities have to be realized by means of wireless communication and in relation with highly constrained devices, our intention is to minimize overheads generated and sustained by our management system on the mobile side. From a business point of view, the quality of service must not be attenuated by administra-

Figure 4. Canonical behavior of a wireless component

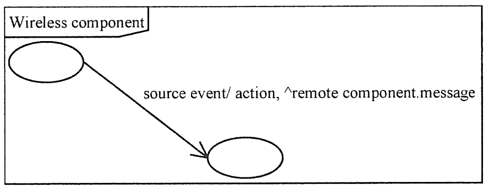


Figure 5. Common communication scraps in wireless environments

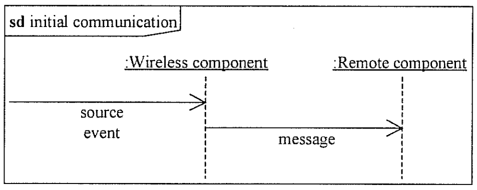
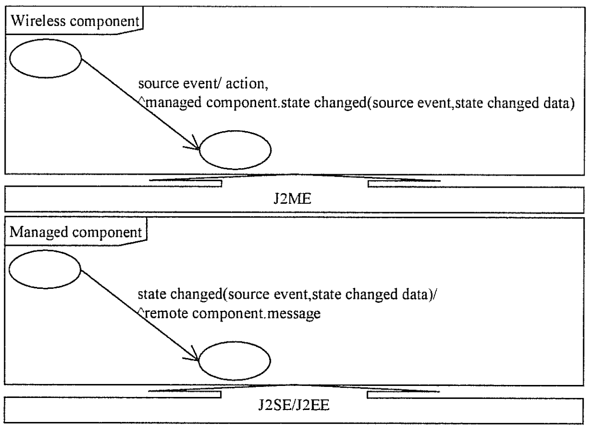


Figure 6. Overview of the technical framework


Wireless component
source event/ action,
^managed component.state changed(source event,state changed data)
J2ME
Managed component
state changed(source event,state changed data)/
^remote component.message
J2SE/J2EE

tion course. We purposefully create replications (caches or images) of managed components in the management system (see Figure 3).

Wireless Component State Machines

Figure 4 depicts the canonical behavior of an ordinary wireless software component. In other words, *source event* corresponds to a typed reaction to an external phenomenon as, for instance, a pressed device button. Numerous actions may be (in order: use of "," for formalizing chronology in Figure 4) launched in response to the processing of *source event*. Most of the actions are the displaying and/or the refreshing of the mobile device's user interface.

Only two actions are drawn in Figure 4: *action* is an internal action (typically, displaying something) in the sense that it does not require any other resources than those available within the modeled device. In contrast to *^remote component.message*, *action* also does not yield any communication. In UML, an instance of the *SendSignalAction* metatype is represented by an expression similar to *^remote component.message*. That is by definition a communication unit towards a remote component in order to create a global collaboration between several components. Such a collaboration corresponds to a strict requirement ("use case" in UML) of the designed wireless environment. This may be summarized by means of a UML Sequence Diagram (see Figure 5).

Mirroring

At this time, components deployed in mobile devices have poor autonomy. Therefore, most collaborations, like the one shown in Figure 5, are recurrent and systematic. We here mean that peer-to-peer communication, nowadays, may not be considered as a technological lock for mobile devices, but transforming such equipment into Internet "points" (i.e., Internet self-contained and autonomous nodes) is slowed down by many factors, such as hardware and platform limits, security constraints, and so forth. For instance, one cannot today run a JMX server on a mobile device for administration purposes.

Starting from this general observation, our technical framework consists of having an intermediate or mirror component, named managed component, between wireless and remote components. Moreover, managed components are exact clones of wireless components (see Figure 3). The difference between wireless components and managed components is, as indicated in their generic name, that these last ones are plugged into management environments. In Figure 6, states machines of each type are specified, as well as a new way for communicating: We review the models in Figures 4 and 5 so that managed components may act as proxies between wireless and remote components.

In addition, since our technical framework is based on Java, we express the need for a J2SE/J2EE architecture in order to have at one's disposal administration facilities. On the other hand, J2ME serves as the software platform for wireless components. The idea is simple: administrating wireless components occurs through the administration of perfect clones.

Communication is thus revised in Figure 6 so that exchanges with remote components are replaced in wireless components by notifications of state changes carrying all of the necessary data which allows managed components to rule their automaton in symbiosis with that of their source. Clearly, the automaton of a wireless component is implemented twice: within the mobile device in which it acts as a driver, and through the automaton of the managed component. It always possesses the same shape in the sense that managed components imitate their sources. Furthermore, Figure 6 shows that managed components serve as delegates between wireless and remote components: *^remote component.message* now appears in managed components' automata instead of in those of wireless components.

While the automaton on the top of Figure 6 supplies the emission of the *state changed* message, the other one, beneath, shows the

Figure 7. Common communication scraps in administrated wireless environments

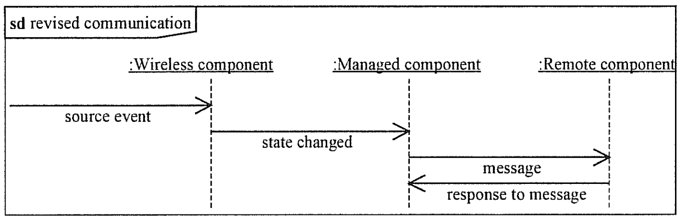
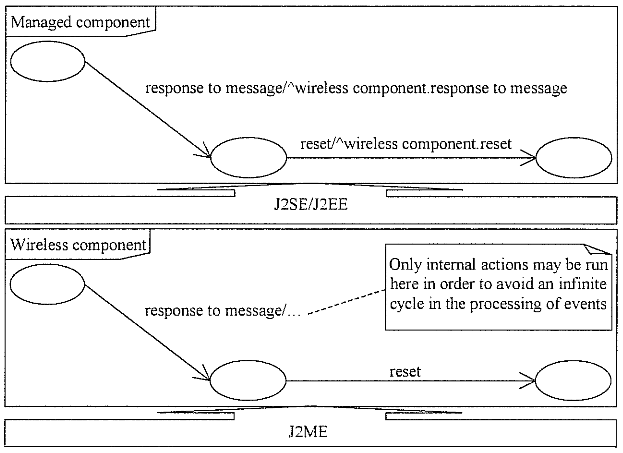


Figure 8. Inverse direction of communication for control purposes


Managed component
response to message/^wireless component.response to message
reset/^wireless component.reset
J2SE/J2EE
Wireless component
response to message/...
Only internal actions may be run here in order to avoid an infinite cycle in the processing of events
reset
J2ME

consequential reception. As a result, both automata are "equivalent," meaning that they are not physically synchronized, but they go through the same lifecycle. The unique problem relating to such an interval is that control activities have to done with care. However, the original communication mechanism of statecharts is broadcast communication (Harel, 1987) that by definition makes no assumption about receivers' status. That is the case here when possible responses go back to wireless components: any state is acceptable. The broadcast communication mechanism is supported by the *PauWare.Statecharts* library.

Control

Managed components, as members of distributed applications, are subject to requests coming from varied components. In Figure 7, the scenario shows that an event called *response to message* must be processed within the managed component's statechart that has previously sent *message*. From a functional point of view, *response to message* is destined to the managed component's source: the wireless component is a replication.

Besides, managed components in essence receive administrative requests (e.g., *reset* in Figure 8) that also have to be propagated to wireless components. As a result, models in Figure 8 sum up both kinds of reaction.

Design Method

The generic micro-architecture specific to our notions of wireless component and managed image may be synthesized by means of a UML Component Diagram (see Figure 9). Wireless components provide two kinds of interfaces. The first one is for local events (*Wireless component local provided interface*) as, for instance, keyboard inputs (*source event* in Figures 4, 5, 6, and 7). The second one is for requests coming from outside (*Wireless component remote provided interface*) that must together be implemented by wireless and managed components (example: *response to mes-*

Figure 9. Assembly pattern between wireless and managed components (UML 2 Component Diagram)

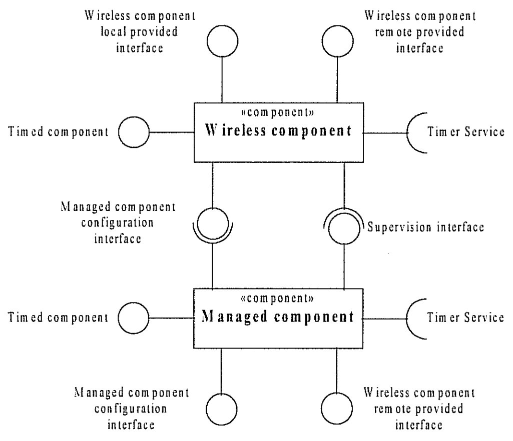

Wireless component local provided interface
Wireless component remote provided interface
«component»
Wireless component
Timed component
Timer Service
Managed component configuration interface
Supervision interface
«component»
Managed component
Timed component
Timer Service
Managed component configuration interface
Wireless component remote provided interface

Figure 10. Detailed basic operations offered in each kind of interfaces from Figure 9

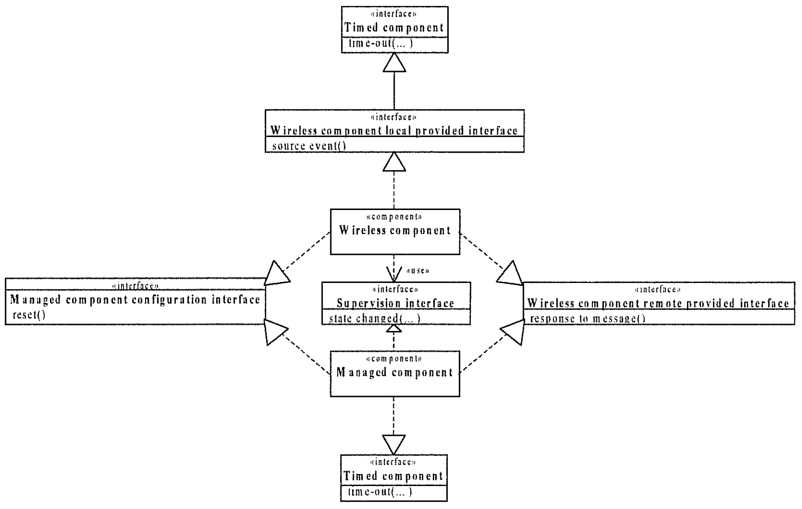


Figure 11. Alternative to Figure 10

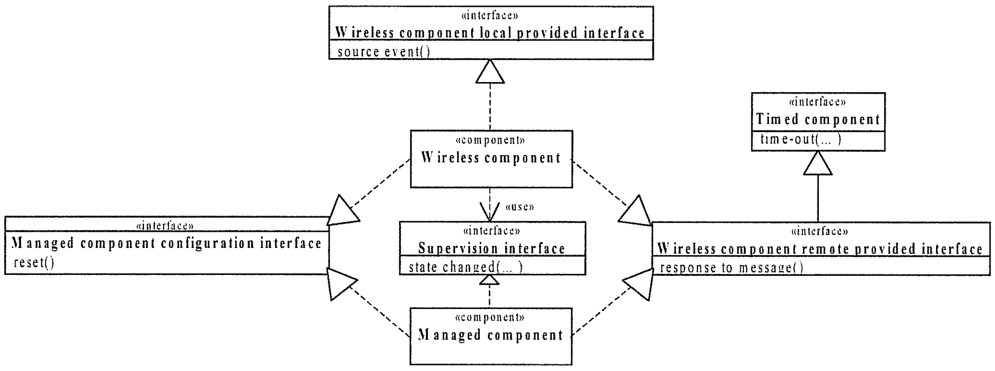
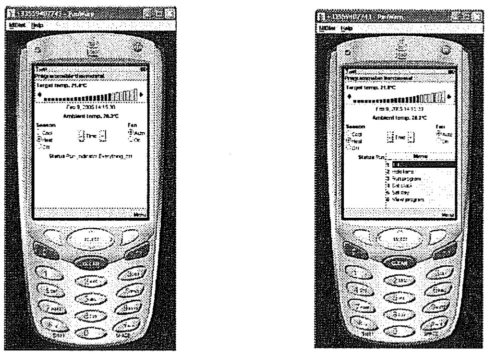


Figure 12. User interface of the home automation device and its possible evolution



sage that is interpreted in both automata in Figure 8).

In the administration context, management operations are assigned to special interfaces, namely *Managed component configuration interface* and *Supervision interface* in Figure 9. Finally, specific incoming events and flows for wireless components may lead to creation of similar phenomena on the managed components' side without any communication. In Figures 9 and 10, we take an example about timer services that are located and thus acquired from different running platforms. This means here that interpreted *time-out* events on the wireless components' side are simulated with the same contexts and constraints on the managed components' side. More generally, this leads to the detailed and competing views in Figures 10 and 11 that raise the problem of control flow segmentation.

In Figure 11, *Timed component* is inherited by *Wireless component remote provided interface* instead of being independently implemented (symbol is white triangle with dotted

Figure 13. Component architecture of the home automation system

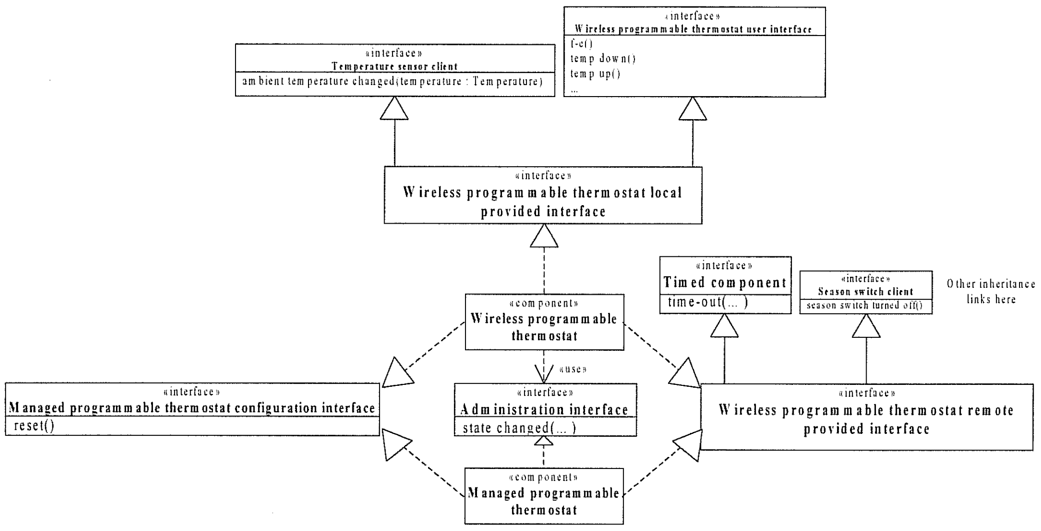
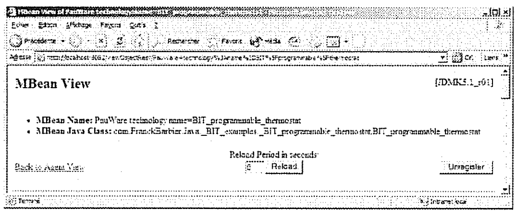

«interface»
Temperature sensor client
ambient temperature changed(temperature : Temperature)
«interface»
Wireless programmable thermostat user interface
f-c()
temp down()
temp up()
...
«interface»
Wireless programmable thermostat local provided interface
«component»
Wireless programmable thermostat
«use»
«interface»
Timed component
time-out(...)
«interface»
Season switch client
season switch turned off()
Other inheritance links here
«interface»
Managed programmable thermostat configuration interface
reset()
«interface»
Administration interface
state changed(...)
«interface»
Wireless programmable thermostat remote provided interface
«component»
Managed programmable thermostat

Figure 14. Web browser screenshot of the management environment


Fichier Edition Affichage Favoris Outils ?
Précédente Rechercher Favoris Média
Adresse
OK Liens

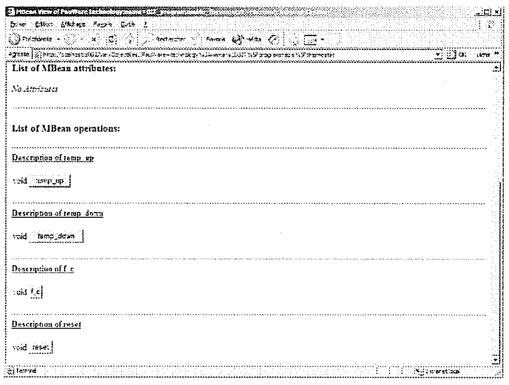
MBean View [JDMK5.1_r01]

- **MBean Name:** PauWare.technology:name=BIT_programmable_thermostat
- **MBean Java Class:** com.FranckBarbier.Java._BIT_examples._BIT_programmable_thermostat.BIT_programmable_thermostat

Back to Agent View

Reload Period in seconds: 0 Reload

Unregister


Fichier Edition Affichage Favoris Outils ?
Précédente Rechercher Favoris Média
Adresse
OK Liens

List of MBean attributes:

No Attributes

List of MBean operations:

Description of temp_up

void temp_up

Description of temp_down

void temp_down

Description of f_c

void f_c

Description of reset

void reset

line in UML) by both wireless and managed components. Note that Figure 9 deliberately makes no assumption on this point. In Figure 11, being thus "included" in *Wireless component remote provided interface*, the *Timed component* interface makes managed components receiving *time-out* events from wireless components instead of using platform-dependent services (i.e., J2SE/J2EE services). So, both implementations of *Timed component* continue to differ, but that of managed components is fixed and unique, accepting *time-out* events coming from wireless components.

CASE STUDY: A HOME AUTOMATION SYSTEM

Figure 12 is the user interface that is an entry for some of the received events (e.g., *temp down*, *temp up*, *f-c*) of the complex statechart appearing in the Appendix. Although being complex, the UML statechart in the Appendix is easily and straightforwardly implemented thanks to the *PauWare.Statecharts* library (see the following code).

Figure 13 shows the chosen component architecture, namely the separation between what is locally received by the *Wireless programmable thermostat* component which implements *Wireless programmable thermostat local provided interface,* and what is received by the *Managed programmable thermostat* component through *Wireless programmable thermostat remote provided interface*.

For instance, temperature sensor events (*ambient temperature changed*) are acquired by *Wireless programmable thermostat* and propagate to *Managed programmable thermostat,* while switching events, such as *season switch turned off* which definitively stops the overall home automation system (see statechart in Appendix), are connected with any port of *Managed programmable thermostat* and embody control commands when delegated to *Wireless programmable thermostat*.

Figure 14 is the final result showing all of the possible operations supported by the *Managed programmable thermostat* component in a Web browser. Visible buttons in windows of Figure 14 simply and straightforwardly map to service implementation for *Managed programmable thermostat*. Here is some Java code illustrating how the *PauWare.Statecharts* library helps implementation:

```
public void f_c() throws Statechart_exception {
   // f-c event (see Appendix and/or popup
   menu in Figure 12)
   _BIT_programmable_thermostat.fires(
   _Ambient_temperature_displaying,_Ambient
   _temperature_displaying);
   _BIT_programmable_thermostat.fires(
   _Target_temperature_displaying,_Target
   _temperature_displaying,true,this,"switch_mode");
   _BIT_programmable_thermostat.fires(
   _Program_target_temperature
   _refreshing,_Program_target_temperature_
   refreshing,true,this,"switch_mode");
   _BIT_programmable_thermostat.run_to
   _completion("f-c");
   }
```

The *run_to_completion* predefined routine conforms to UML 2 executability rules and, consequently, lasts through stable and safe automaton contexts. Management operations may thus occur as the *reset* user-defined function that may look like:

```
   public void reset() throws Statechart
   _exception { // configuration service
        to_state("Operate"); // Operate may
        be seen within the statechart in Appendix
   }
```

CONCLUSION AND FUTURE DIRECTIONS

This chapter highlights the great need for administration of mobile environments, including the management of the collaboration between software components deployed in these environments and software components deployed in server-side tiers. Numerous hardware/software barriers preclude having full administration capabilities on wireless sides only. Furthermore, even in traditional management environments, logic and rationale for expressing supervision and control operations do not exist. For instance, JMX supplies a standard infrastructure but does not explain how to design manageable components.

The chapter shows that executable modeling, supported by the statecharts' reputable formalism associated with a dedicated library, both allow the methodical design of components for administration. The proposed approach is illustrated by a programmable thermostat wireless component.

REFERENCES

Barbier, F. (2005). COTS component testing through built-in test. In S. Beydeda & V. Gruhn (Eds.), *Testing commercial-off-the-shelf components and systems* (p. 55-70). Berlin: Springer-Verlag.

Bloch, C., & Wagner, A. (2003). *MIDP 2.0 style guide for the Java 2 platform* (micro ed.). San Francisco: Addison-Wesley.

Bock, C. (2004). UML 2 composition model. *Journal of Object Technology, 3*(10), 47-73.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8,* 231-274.

Horrocks, I. (1999). *Constructing the user interface with statecharts.* San Francisco: Addison-Wesley.

Kreger, H., Harold, W., & Williamson, L. (2003). *Java and JMX—building manageable systems.* San Francisco: Addison-Wesley.

Mellor, S., & Balcer, S. (2002). *Executable UML—a foundation for model-driven architecture.* San Francisco: Addison-Wesley.

National Coordination Office for Information Technology Research and Development. (2001). *High confidence software and systems research needs.* Arlington, VA: NCO/ITRD.

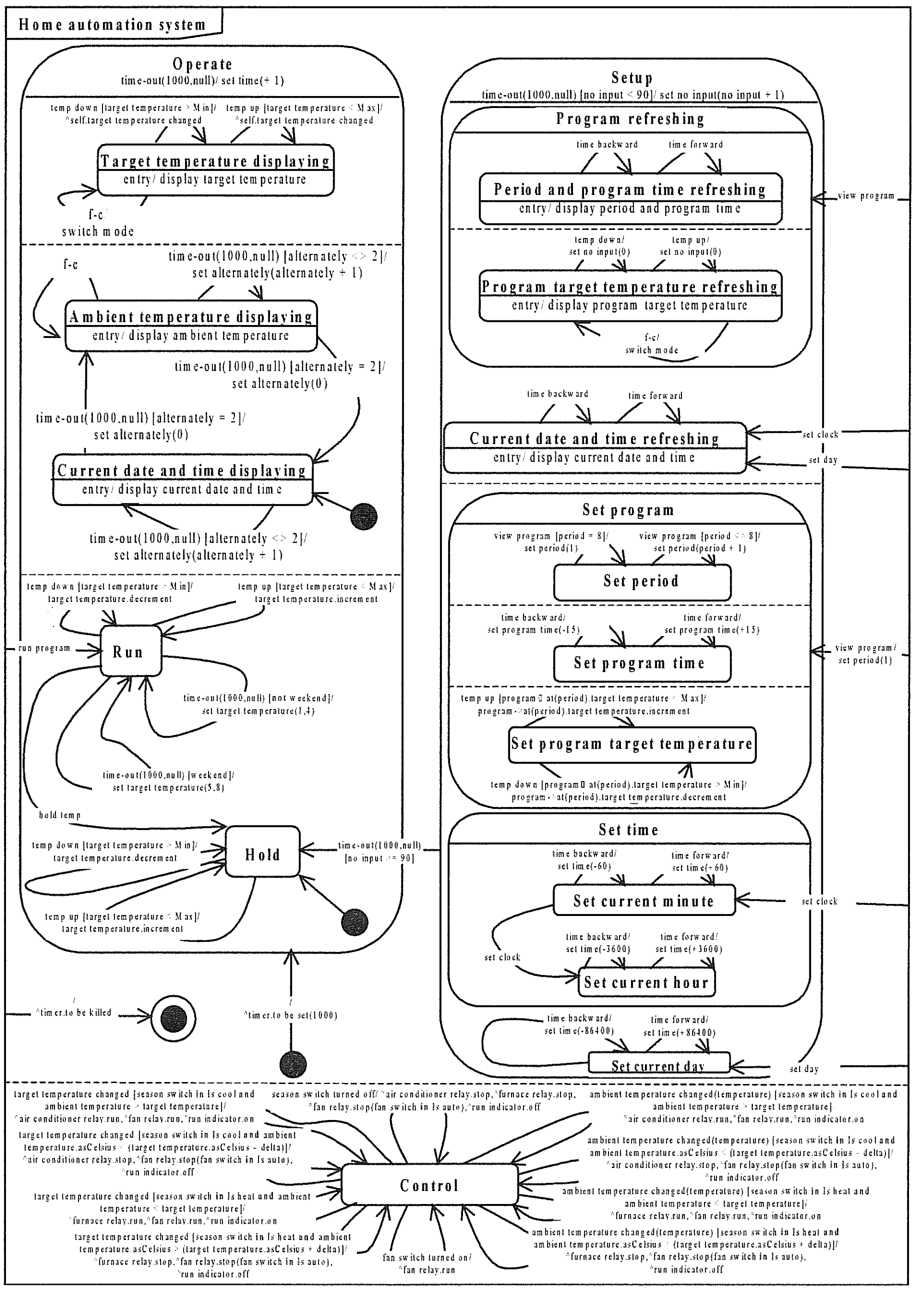
Object Management Group. (2003). *UML 2.0 infrastructure specification.* Needham, MA: OMG.

Object Management Group (2003). *UML 2.0 superstructure specification.* Needham, MA: OMG.

Sun Microsystems. (2002). *Java management extensions instrumentation and agent specification, v1.2.* Santa Clara, CA: Sun Microsystems.

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component software—beyond object-oriented programming* (2nd ed.). San Francisco: Addison-Wesley.

APPENDIX: STATECHART OF THE PROGRAMMABLE THERMOSTAT WIRELESS COMPONENT


Home automation system
Operate
time-out(1000,null)/ set time(+ 1)
temp down [target temperature > Min]/ ^self.target temperature changed
temp up [target temperature < Max]/ ^self.target temperature changed
Target temperature displaying
entry/ display target temperature
f-c/ switch mode
f-c
time-out(1000,null) [alternately <> 2]/ set alternately(alternately + 1)
Ambient temperature displaying
entry/ display ambient temperature
time-out(1000,null) [alternately = 2]/ set alternately(0)
time-out(1000,null) [alternately = 2]/ set alternately(0)
Current date and time displaying
entry/ display current date and time
time-out(1000,null) [alternately <> 2]/ set alternately(alternately + 1)
temp down [target temperature > Min]/ target temperature.decrement
temp up [target temperature < Max]/ target temperature.increment
run program
Run
time-out(1000,null) [not weekend]/ set target temperature(1,4)
time-out(1000,null) [weekend]/ set target temperature(5,8)
hold temp
temp down [target temperature > Min]/ target temperature.decrement
Hold
time-out(1000,null) [no input >= 90]
temp up [target temperature < Max]/ target temperature.increment
/ ^timer.to be killed
/ ^timer.to be set(1000)
Setup
time-out(1000,null) [no input < 90]/ set no input(no input + 1)
Program refreshing
time backward
time forward
Period and program time refreshing
entry/ display period and program time
view program
temp down/ set no input(0)
temp up/ set no input(0)
Program target temperature refreshing
entry/ display program target temperature
f-c/ switch mode
time backward
time forward
Current date and time refreshing
entry/ display current date and time
set clock
set day
Set program
view program [period = 8]/ set period(1)
view program [period <> 8]/ set period(period + 1)
Set period
time backward/ set program time(-15)
time forward/ set program time(+15)
Set program time
view program/ set period(1)
temp up [program□ at(period).target temperature < Max]/ program->at(period).target temperature.increment
Set program target temperature
temp down [program□ at(period).target temperature > Min]/ program->at(period).target temperature.decrement
Set time
time backward/ set time(-60)
time forward/ set time(+60)
Set current minute
set clock
time backward/ set time(-3600)
time forward/ set time(+3600)
set clock
Set current hour
time backward/ set time(-86400)
time forward/ set time(+86400)
Set current day
set day
target temperature changed [season switch in Is cool and ambient temperature > target temperature]/ ^air conditioner relay.run,^fan relay.run,^run indicator.on
season switch turned off/ ^air conditioner relay.stop,^furnace relay.stop, ^fan relay.stop(fan switch in Is auto),^run indicator.off
ambient temperature changed(temperature) [season switch in Is cool and ambient temperature > target temperature]/ ^air conditioner relay.run,^fan relay.run,^run indicator.on
target temperature changed [season switch in Is cool and ambient temperature.asCelsius < (target temperature.asCelsius - delta)]/ ^air conditioner relay.stop,^fan relay.stop(fan switch in Is auto), ^run indicator.off
ambient temperature changed(temperature) [season switch in Is cool and ambient temperature.asCelsius < (target temperature.asCelsius - delta)]/ ^air conditioner relay.stop,^fan relay.stop(fan switch in Is auto), ^run indicator.off
Control
target temperature changed [season switch in Is heat and ambient temperature < target temperature]/ ^furnace relay.run,^fan relay.run,^run indicator.on
ambient temperature changed(temperature) [season switch in Is heat and ambient temperature < target temperature]/ ^furnace relay.run,^fan relay.run,^run indicator.on
target temperature changed [season switch in Is heat and ambient temperature.asCelsius > (target temperature.asCelsius + delta)]/ ^furnace relay.stop,^fan relay.stop(fan switch in Is auto), ^run indicator.off
fan switch turned on/ ^fan relay.run
ambient temperature changed(temperature) [season switch in Is heat and ambient temperature.asCelsius > (target temperature.asCelsius + delta)]/ ^furnace relay.stop,^fan relay.stop(fan switch in Is auto), ^run indicator.off