Autonomic Management of Component-Based Embedded Software

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
*Email: fabien.romeo@univ-pau.fr
†Email: franck.barbier@franckbarbier.com
‡Email: jean-michel.bruel@univ-pau.fr

***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.

II. 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 func-

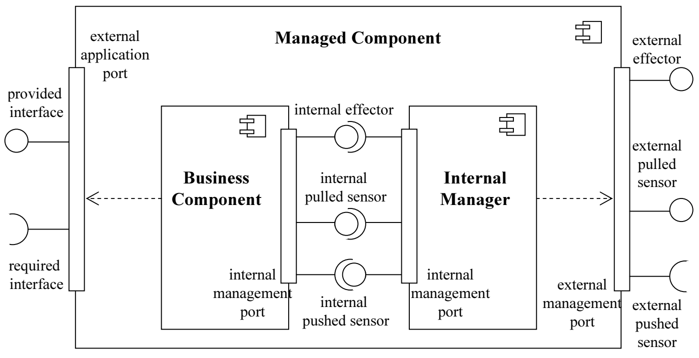

Fig. 1. Managed Component Architecture

tionalities, *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 IV).

III. 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 *ˆself.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.

IV. External Management of Components

In the previous section we have seen that the internal manager is responsible for the direct monitoring and controlling 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 twofold. 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.

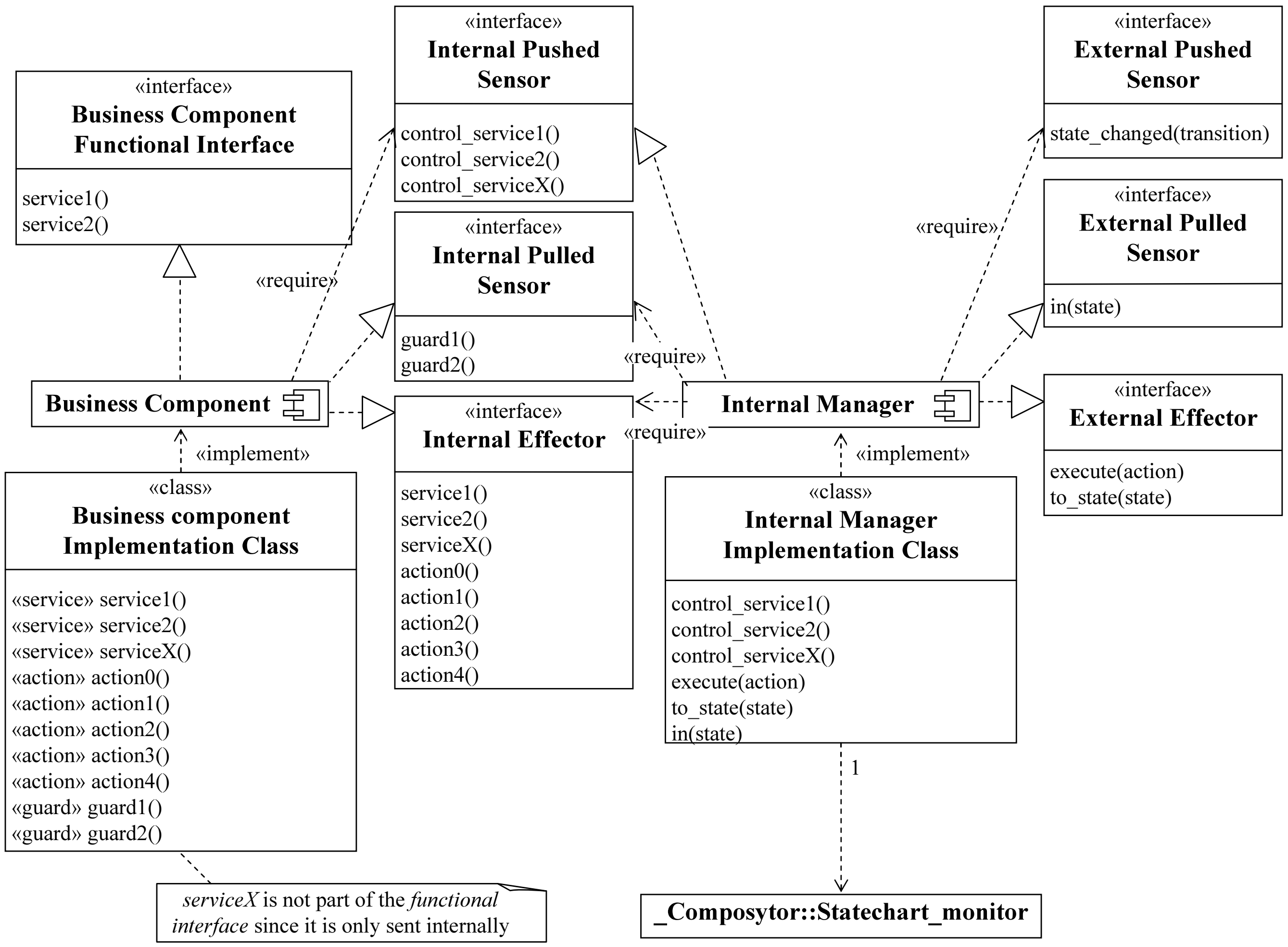

Fig. 2. Managed Component Detailed Architecture

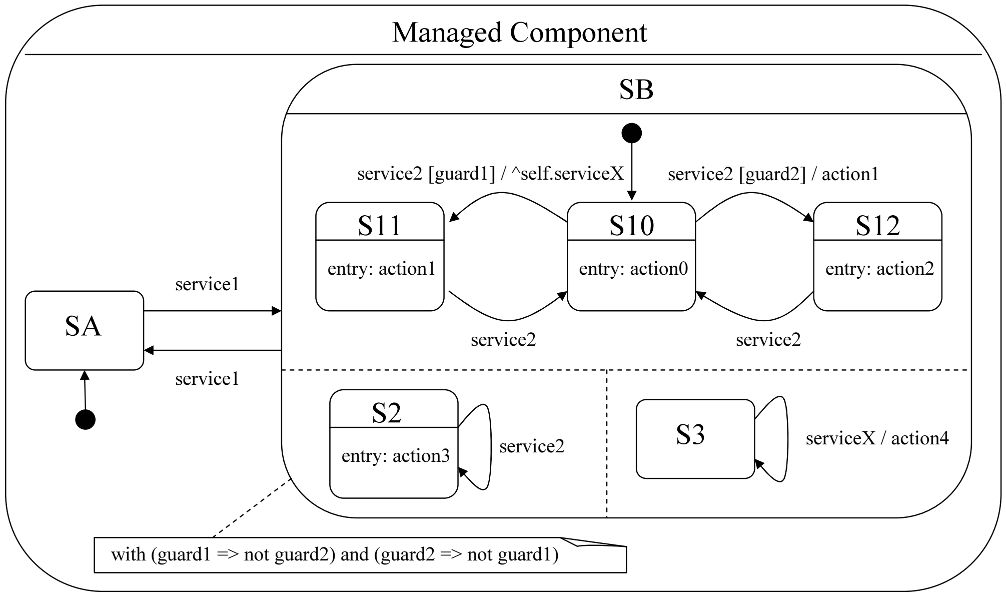

Fig. 3. Managed Component Behavior

In managed component internals, the data we managed are events and low-level states which are captured at runtime by the state machine as *abstract* current states and fired transitions (as shown in section III). Only these higher-level data will be managed through external sensors and effectors. This provides an enhanced vision of the component's business logic without decreasing the management possibilities. Indeed, 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.

V. 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, the remote management system can efficiently monitor and control them.

We have validated our approach by a prototype running on real devices and implemented for the Java Platforms. This infrastructure is named WMX[1], which stands for Wireless Management Extensions. 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.

Quantitatively speaking, WMX fits into a 47.4 Ko jar bundle for the embedded side and our benchmarks reveal only 50.73% of execution time overhead compared to an output on a simple log console (average time is 38.91 μs against 25.82 μs per state change on our test system).

We are also currently working on higher 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.

[1]available at `http://www.univ-pau.fr/~fromeo/wmx`
[2]available at `http://www.pauware.com`

References

[1] K. C. Wallnau, "Volume III: A Technology for Predictable Assembly from Certifi able Components," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, Tech. Rep., april 2003.

[2] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[3] M. Winter, T. Genssler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arvalo, P. Mller, C. Stich, and B. Schnhage, "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] H. Cervantes and R. S. Hall, "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, September 2000.

[5] M. Desertot, H. Cervantes, and D. Donsez, "FROGi: Fractal components deployment over OSGi," in *5th International Symposium on Software Composition SC'06*, Vienna, Austria, March 2006.

[6] I. Crnkovic, "Component-based Software Engineering for Embedded Systems," in *International Conference on Software engineering, ICSE'05*. St. Luis, USA: ACM, May 2005.

[7] A. Möller, J. Fröberg, and M. Nolin, "Industrial Requirements on Component Technologies for Embedded Systems," in *International Symposium on Component-based Software Engineering (CBSE7)*. Edinburgh, Scotland: Springer Verlag, May 2004. [Online]. Available: http://www.mrtc.mdh.se/index.phtml?choice=publications& id=0687

[8] T. Kindberg and A. Fox, "System Software for Ubiquitous Computing," *IEEE Pervasive Computing*, vol. 1, no. 1, pp. 70–81, 2002.

[9] J. Kephart and D. Chess, "The Vision of Autonomic Computing," in *Computer Magazine*. IEEE Computer Society, 2003, vol. 36, pp. 41–50.

[10] H. Liu and M. Parashar, "A component based programming framework for autonomic applications," in *International Conference on Autonomic Computing*, New York, NY, 2004.

[11] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

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

[13] L. E. Buzato, "Management of Object-Oriented Action-Based Distributed Programs," Ph.D. dissertation, University of Newcastle upon Tyne, 1994.

[14] H. Kopetz and N. Suri, "Compositional design of RT systems: A conceptual basis for specifi cation of linking interfaces," in *6th IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Hokkaido, Japan, may 2003.

[15] K. K. Lau, P. V. Elizondo, and Z. Wang, "Exogenous Connectors for Software Components," in *Eighth International SIGSOFT Symposium on Component-based Software Engineering*. Springer Verlag, january 2005.

[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.

[17] J.-P. Martin-Flatin, "Push vs. Pull in Web-Based Network Management," in *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, Boston, MA, May 1999, pp. 3–18.