Model-based Run-Time Adaptation for Statechart-based Components

Xabier Elkorobarrutia[1], Franck Barbier[2], and Goiuria Sagardui[1]

[1] Mondragon Goi Eskola Politeknikoa
Loramendi 4, 20500 Mondragon, Spain
xelkorobarrutia, gsagardui@eps.mondragon.edu
[2] Université de Pau et des Pays de l'Adour
Avenue de l'Université B.P. 1155, 64013 Pau Cedex, France
Franck.Barbier@FranckBarbier.com

Abstract. This article deals with a mechanism for modifying run-time behavior of statechart-based software components. The main target of the presented approach are reactive and control systems, where the main source of failures and uncertainties are due to *environmental faults* and situations arisen from the numerous different working conditions that a system must deal with. This leads to complex situations that if not handled at an early stage, generate stop and maintenance cost, even risks or dangers.
The proposed adaptation mechanism is applied to each individual component. First, we model the behavior for normal or standard conditions by means of statecharts, and then, any variation will be modeled as a modification of the initial model. Upon the detection of environmental variations, those model modifications will be launched at run-time. This is the foundation of this paper's contribution to manage adaptation of components.

Key words: Reconfiguration, statechart, run-time adaptation.

1 Introduction

Adaptation means the ability to cope with new situations. In biological and sociological systems, this is obtained through structural or behavioral changes. The main stream in self* systems attempts to give a system the desired properties through structure changes and component configuration [1]. The basic support for that are component platforms: by means of replacing, replicating, relocating or/and configuring components and assemblies, systems can maintain at lest some minimum level of the desired QoS.

This approach mainly deals with factors that are external to specific applications' logic: hardware failures, external attacks, excessive workloads, etc. However, as mentioned in [2], systems rarely are complete in all senses: specifications are incomplete, the design of the system is in evolution, the internals of some third party components are not well known, etc. When focusing on control

systems, we also should consider those uncertainties originated for operating in unstable environments. Typically, sensors and actuators are the main source of failures inside the control system itself. Additionally, it is difficult to determine how the environment affects the system, and in particular to the sensors.

Those uncertainties are specific to each application and traditionally are managed in a specific way. As noticed in [3], the adaptation mechanism implemented in embedded systems has evolved and will probably evolve in the following way. Indeed, in a first stage, adaptation was not considered. At a second stage, it had not been separated from the rest of the functionalities. At a third stage, where we probably are nowadays, we consciously consider it but there is no proven engineering practice to cope with it in a systematic way. And at a final stage, this engineering practice will arise. Moreover, such adaptation mechanisms could be a complementary technique to traditional redundancy-based fault-tolerance approaches, even a cheaper one.

Reconfigurability provides the foundation upon which autonomic systems can ***adapt*** to their changing environments. This is useful for dynamically optimizing system functionality based upon the observed execution profile or for recovering from errors and failures without human intervention [4]. The main stream in self* communities consider components as the basic blocks for reconfiguration. This enables to deal with systems from an overall architecture perspective. Some few works operate at the virtual machine level in managed environments like JAVA or CLR (for example, [5]). Whenever the case, all use some kind of model at run-time. They also act on systems in terms of the concepts of the meta-model in which those models are based upon.

However, there are other kinds of models that are only used at design-time. In case of software components modeled by means of statecharts, such models are used as specifications or descriptions. There are a lot of design patterns in the literature and many tools in the market exist that make us easier their transformation into code. Most of the times, the model gets lost at run time. This implies that if a system/component can be operating in many different but not simultaneous environmental situations or working conditions (***working mode***), the developer must cope with all possible situations. This increases the complexity of the behavioral model of the component.

This paper focuses on software components which have been specified mainly by means of statecharts and whose behavior, upon the reception of any event, is governed by such statechart. In some cases, a modification of the control part (the statechart) of the component is a suitable strategy as an adaptation mechanism to different working modes. First, we propose a mechanism for modeling adaptation of software components as model changes and then, we describe a framework that supports and assists the development of such components. The chosen approach is close to agent theory. In some cases, local algorithms are viewed as suitable for adaptation purposes.

In section 2, it is presented a concise and representative problem that illustrates how the adaptation mechanism described in this paper is constructed. Section 3 describes a required support to implement that mechanism. Section

4 puts together the method and the support treated in sections 2 and 3. Finally, section 5 copes with the applicability of the approach, its extensions and improvements.

2 Behavior Changes Modeling

Let us consider a software component in charge of controlling the temperature of a room by means of a heater and a cooler. Having omitted some details, its control part is illustrated in Fig. 1. Although we present this component in isolation, it is part of a broader system in charge of the air-conditioning/heating and energy management system of an entire building. This component, when receiving a command that specifies a time interval and a desired temperature, switches to the controlling state. Consequently, it switches among its various substates depending on the real temperature until the specified time interval expires.

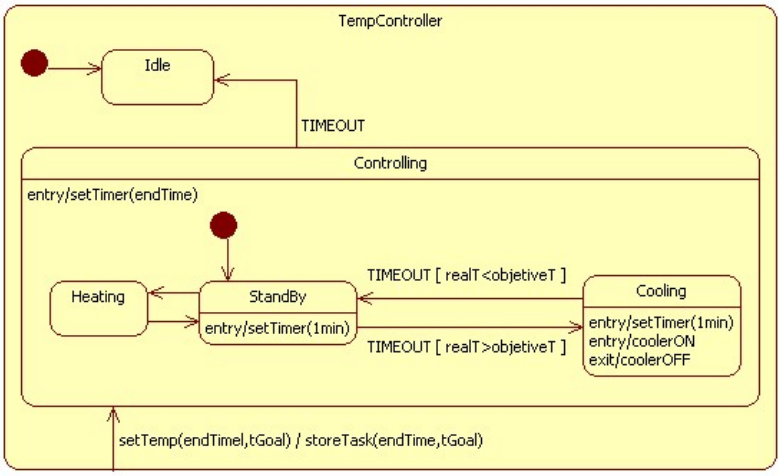


Fig. 1. The Control of the behavior of a room temperature controller.

Let us now assume another variant of such a control element. Suppose that in the room there is also a presence detector. In this context, the behavior of the controller can be adapted, so that whenever the presence detector detects that nobody is in the room, the controller must stop the control of the temperature. It is resumed when presence is detected. This new behavior is depicted in Fig. 2.

To imagine a more complicated behavior, *suppose that the presence detector can become faulty and in this case, we want the controller to switch between*

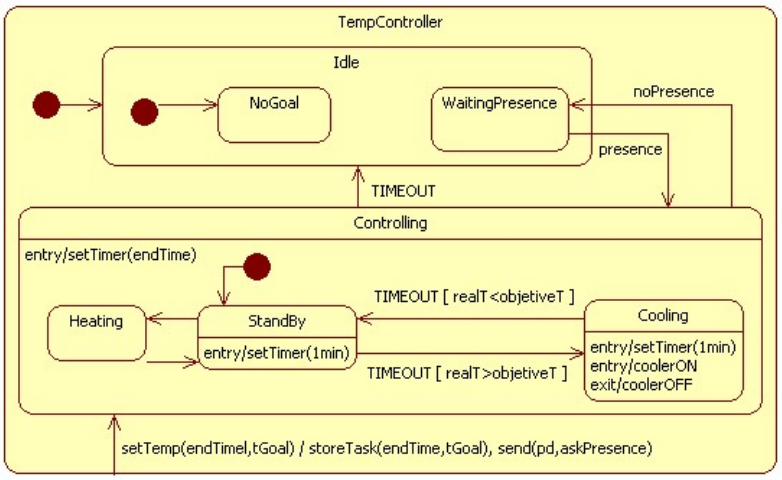


Fig. 2. Extended room temperature controller designed for working in collaboration with a presence detector.

both working modes at run-time. Regarding those specifications, we need some additional mechanisms to made possible such an adaptation.

Traditionally, this kind of issues have been solved in an application-specific way. Both behaviors, although being exclusive, are mixed in one unique model that implicitly supports many orthogonal working modes. That does not mean that the developer is unaware of that fact. But the working mode changes are hardwired with other functional aspects. Complexity of the components grows due to extra guards or sentences like: `{if(mode=MODE1) then ...}`. When developing the component, we need to cope with all working modes simultaneously. Instead, if we could model both working modes in a separate way and if we had the necessary support to switch them at run-time, the developer could use it as an explicit adaptation mechanism based on redundancy of control algorithms. This mechanism should be coupled with an infrastructure for detecting such conditions that may launch at run-time a model change in each particular component.

3 Support for Statecharts Based Components

As previously mentioned, we are interested in systems whose software components are modeled mainly by means of statecharts. For implementation of finite state machines and statecharts, there are a lot of design-patterns [6] and commercial tools. Most of them are suit to deal with plain state machines and fewer of them cope with statecharts. There are several criteria that makes one pattern preferable to others depending of the context in which they are applied. This is why we can not talk about an optimum implementation. The majority of them

aim at transforming the model to implementation in a systematic way. Once the code is obtained, the model is forgotten.

Few approaches maintain the statechart or finite state machine model at run-time. In [7], an approach is proposed for dynamic change of the behavior associated with each particular state. However, there is no possibility for effecting this dynamic change with transitions or other modeling elements and it is only applicable to plain state-machines. In [8] a reflective structure is implemented inside each software component which is modifiable at run-time. But it is not used for run-time reconfiguration purposes and it needs some extensions in order to gain an adaptation ability.

The approach presented here will permit to develop such software components in a model driven style of development, and with the aid of a framework, the statechart model will remain in a reflective structure enabling its change at run-time. For this purpose, we have developed an experimental framework called ISCART in JAVA ([9]). Fig. 3 shows the overall structure of such software components.

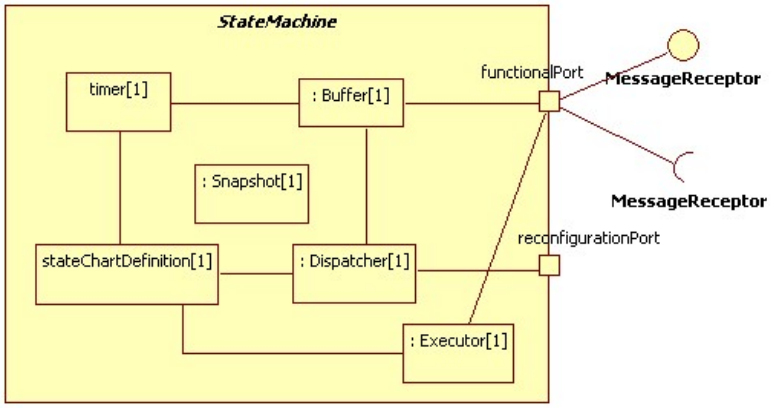


Fig. 3. Overall structure of the proposed software components.

Beforehand, we must delimit what model elements will be reflected at run-time. In order to be applicable to any software component, it has been decided to separate those aspects pertaining to statechart language from those pertaining to each specific application. For example, if upon the reception of an event E we have a transition from state S_1 to S_2, guarded by a condition C and has associated the action A, all those elements will be reflected by the software component except the internal implementation of the guard evaluation and the action execution. For those last two elements the reflection infrastructure will have a reference of them. Let briefly describe the elements that constitute the software components (Fig. 3) constructed by our framework:

- The event reception will be implemented through messages. The communication model for our components is based on asynchronous messaging with reception acknowledgement. For that issue we have the `FunctionalPort` and `Buffer` parts.
- There is a `Timer` element that offers the timeout facility.
- The main part, `statechartDefinition`, is a composite structure that reflects the statechart model of the component. The `snapshot` part is a global repository of the component that references the states that are active and mantains the message it is being processed in each instant of execution. This last feature avoids many parameter passing problems among different parts.
- `executor` is an object that maintains all the *"pseudovariables"* of the component (counters, references, ...). Some of its methods are responsible for the executing actions and evaluating guards.
- `scheduler` is the element that upon the the reception of a message, interrogates the statechart part to know which sequence of actions should fire. Thus, the executor part is needed for evaluation of the needed guards. Afterwards, it will direct the `executor` to actually execute the list of actions.

It is not our objective to enter into each detail. The way to accomplish the development of such components is through the support of a framework. Fig. 4 shows some classes offered by such framework. The developer must construct the statechart model in a programmatic way specifying state hierarchy, transitions and the rest of the elements that form the model. At initialization time, this results in a complex structure that reflects the initial model. Besides, he must supply the executor class whose methods will be invoked whenever a guard must be evaluated or an action must be executed.

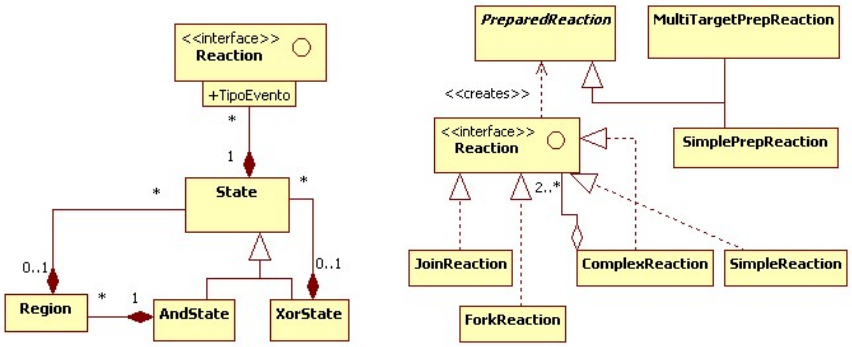


Fig. 4. Main classes whose instances will form the `statechartDefinition` structure.

Whenever the component receives a message, it must first determine which actions should fire. In statechart formalism, compared to their plain state machines counterparts, each transition really is a *family of transitions* . For example, consider in Fig. 2 the transition fired by the `TIMEOUT` event from state

`Controlling` to `Iddle`. In fact, this transition is a factorization of 3 transitions going out from each subestate of `Controlling` towards `Iddle`. The situations aggravates if we employ pseudoestates as *"History"* or *"DeepHistory"* and we have multiple parallel regions.

As shown in Fig. 5, upon the reception of a message, our framework determines the exact reaction in the following way: (1) depending of the current active states, ask for the transitions that may be fired and whose guards permit it. (2) Doing so, the estates of the hierarchy that may be entered are marked and an instance of `PreparedReaction` is created at run-time as a particularization of the transition it becomes from. (4) Finally, the reaction associated to the received message is fired.

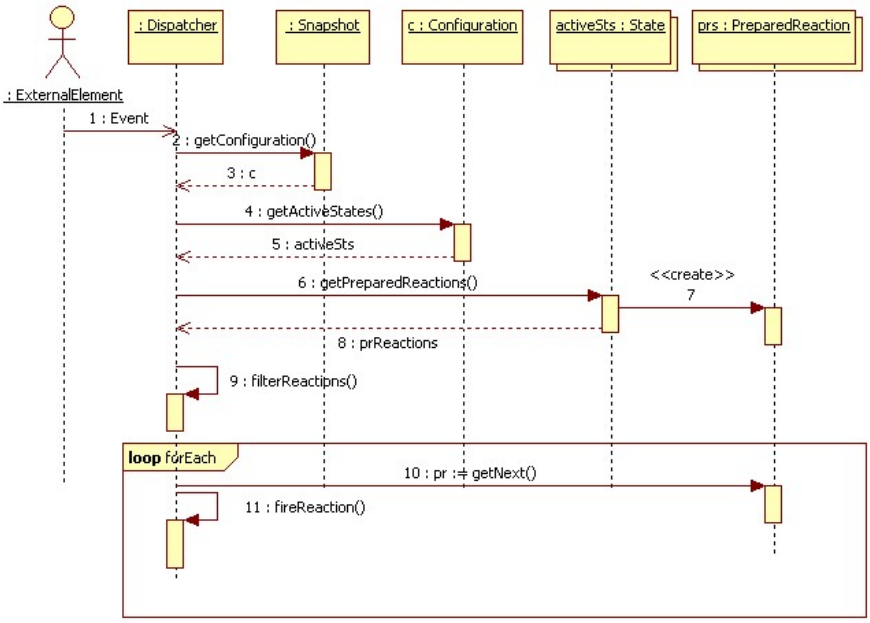


Fig. 5. Determinarion of the reactions to be fired.

Although lacking many details, this section shows the ability to change the control model at run-time. It suffices to modify the `StatechartDefinition` part. What remains now is how to made such changes in a structured way.

4 Behavior Changes at Run-Time

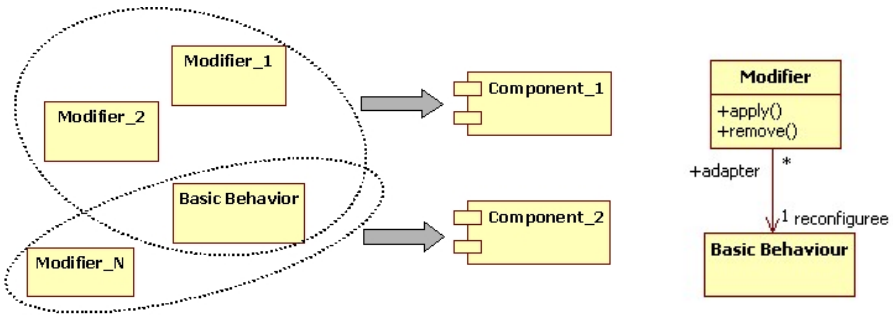
Let us return to the temperature controller example of section 2. Remember that we have specified a behavior change depending on the correct working of the presence detector. We can cope with it in at lest three different ways:

- Implement all working modes hardwiring them and all the rest of functionality (it does not exclude design patterns or language idioms for helping in that duty)
- Develop two alternative components, each of them suitable for one set of circumstances and let an external infrastructure to swap between them.
- Build inside the component a mechanism to swap behavior.

The better suitability of any of them will depend on many factors, e.g. the number of working mode switches, the complexity of coping simultaneously with all them, the facilities offered by and restrictions of a component platform we are considering, etc. Far away of being exclusive, they can complement each other. Particularly we are focusing on the third one.

Let us assume that at development time we have detected various different working modes. How and when must we effect those changes at run-time? In Fig. 3 there is an element called `ReconfigurationPort` that could allow to perform model changes externally not previously programmed. But it only has been depicted to illustrate this capability. The selected approach is described next:

1. The developer team defines a behavior for a software component in normal operation conditions.
2. For each anomalous/different working conditions, the behavior of the component will be modeled as a variation of such first model. Such variations will be called `Modifiers`.
3. The software component will be packaged with the first behavior along with those variations as illustrated in Fig. 6(a)



(a) Software components packaged with multiple modifiers. (b) Modifiers' superclass.

Fig. 6. Each statechart based software component is provided with a basic behavior and a set of modifiers

4. By provision of some detection mechanism, upon the detection of some *"circumstantial change"*, a variation of the actual statechart model will be launched.

Point 4 deserves an explanation of how we have implemented it. As we have preciously mentioned, the part of each software component that reflects the control aspects is a complex structure that is constructed at initialization time. Listing 1.1 shows some code snippets of how the control part of the software component is constructed by means of the interface that out framework offers. If it is done at initialization time, it means that it have been done dynamically. Now, if we want to modify such control part in any instant of execution, we only need to launch similar sentences that modify the structure that reflects the statechart.

Listing 1.1. Initialization(partial) of the temperature controller of Fig. 1

```
protected void initStructure()
{
  sRoot=new XorState("sRoot");
  rootState=sRoot;
  sIdle=new XorState("Idle");
  sRoot.addInitialState(sIdle);
  sControlling=new XorState("Controlling");
  sRoot.addState(sControlling);
  sStandBy=new XorState("StandBy");
  sStandBy.setTimer(60*1000);//1min
  ........
}

protected void initBehaviour()
{
  storeTask=new MethodInvocation("storeTask");
  rStoreTask=new SimpleReaction(sControlling,null,
                                storeTask,"StoreTask");
  sRoot.addReaction(EvSetTemperature.class, rStoreTask);
  .....
}
```

But many questions arises in that issue. Are we going to permit all kind of changes or are we going to restrict them? Is it possible to eliminate an state which is active at the moment of launching the modification? Could we need some mechanism to go backward in the execution history for helping those changes be consistent? Can various modifiers be applied simultaneously? etc. We have not addressed these questions and have only provide a facility for effecting run-time changes. The responsibility of such questions is delegated to the developer.

As shown in fig. 6(b) there is a superclass for each modifier that aims at effecting the changes in a proper way. Their interface is limited by such a superclass and has the unique possibilities of applying a modification or restoring the changes it has made. The developer can launch them as any other action

in response to the reception of an event. Of course, for this to make sense, that event will normally be a kind of notification of some circumstantial change. This mechanism differs from component swapping in that we do not need an external infrastructure to change behavior in a systematic way.

5 Conclusions and Future Work

We have presented an approach based on a framework for effecting in statechart based software components model changes at run-time, i.e. behavior changes. Thanks to such a framework, the mechanism is built inside the component and does not need any external infrastructure. Anyway, it must be complemented with an environment sensing mechanism that guides those behavior changes. Ultimately, this run-time behavior modification can be used as an adaptation mechanism.

But even having a framework that support the developer in run-time behavior changes, there are still some open issues as how to limit those changes in order to avoid semantic misunderstandings. Besides that, we have applied exclusively with statecharts. We did not explore other behavior description artifacts, e.g. activity diagrams as a basis for behavior reconfiguration, and ultimately adaptation mechanism.

References

1. Kramer, Jeff Magee, Jeff: Self-Managed Systems: an Architectural Challenge,In: FOSE '07, Future of Software Engineering, IEEE Computer Society (2007)
2. Koopman, Philip: Elements of the Self-Healing Problem Space. In: Workshop on Software Architectures for Dependable Systems(WADS2003) (2003)
3. Adler, Rasmus, Schneider, Daniel Trapp, Mario: Development of Save and Reliable Embedded Systems Using Dynamic Adaptation. In: M-ADAPT Model Driven Software Adaptation, Berlin (2007)
4. Whisnant, K., Kalbarczyk, Z. T., Iyer R. K.: A System Model for Dynamically Reconfigurable Software. IBM Systems Journal, vol. 42, no. 1, 45 - 59, (Jan 2003).
5. Fuad, M. Muztaba, Deb, Debzani, Oudshoorn, Michael J.: Adding Self-Healing Capabilities into Legacy Object Oriented Applications. In: International Conference on Autonomic and Autonomous Systems ICAS 06 (jul 2006).
6. Adamczyk, P.: The anthology of the finite state machine design patterns. In: The 10th Conference on Pattern Languages of Programs (2003)
7. Ferreira, L.L., Rubira, C.M. . Reflective design patterns to implement fault tolerance. Workshop on Reflective Programming in C++ and Java, OOPSLA98 (1998).
8. Barbier, F. Mde-based design and implementation of autonomic software components. International Conference on Cognitive Informatics, ICCI (2006).
9. Elkorobarrutia, X. Muxika, M. Sagardui, G. Barbier, F. Aretxandieta, X. A Framework for Statechart Based Component Reconfiguration. In: Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems, EASe (2008)