A Framework for Statechart Based Component Reconfiguration

Xabier Elkorobarrutia, Mikel Muxika, Goiuria Sagardui
Mondragon Goi Eskola Politeknikoa
Informatika Saila
Loramendi 4, 20500 Arrasate, Spain
xelkorobarrutia,mmuxika,gsagardui@eps.mondragon.edu

Franck Barbier
Universite de Pau et des Pays de l'Adour
Laboratoire de Informatique
Pau, France
Franck.Barbier@FranckBarbier.com

Xabier Aretxandieta
Ulma Handling Systems
20560 Oñati, Spain
xaretxandieta@manutencion.ulma.es

Abstract

This article describes a reconfiguration mechanism for statechart-based software components and presents a framework that supports it. The reconfiguration capability that each component acquires can be used as a a mechanism for self-healing and better adapting the component to environmental condition variations. The latter can also be considered as a support for coping with incomplete or bad specifications due to the lack of exact knowledge of the environment. It will also be shown that it can be used to easier resolve composition issues when creating a system by means of component-composition.

This framework helps creating statechart based components that reside in the middle ground between a blackbox component and an statechart interpreter. In addition to supporting a Model Driven development style, the framework creates a reflective architecture of the component without any involvement from the developer. This reflectiveness adds the ability to modify the component's statechart model at run-time and can be used as a basis for a self healing mechanism.

1. Introduction

Reconfigurability provides the foundation upon which autonomic systems can adapt to their changing environment or to recover from errors and failures without human intervention [20]. Reconfiguration mechanisms are the means by which we can make the software adapt to varying environmental conditions, incorrect hardware functioning, optimization issues, etc. It groups a lot of different techniques with different purposes.

When talking about reconfiguration at system level most of times it is assumed that we have enough resources and execution environments like J2EE that offer facilities to restore a particular application, to relocate it, to come back to an older version, ... But if we move to embedded systems there is no such resource availability nor such execution platforms. That means that any reconfiguration option must be built-in within the application itself. In [2] there is an observation of the stages that embedded systems are going through with respect to adaptation: in a first stage there is no kind of dynamic adaptation, in a second one it is undistinguishable from functional aspects, in a third stage it is explicitly considered but there is no adaptation engineering, and in a final stage, which we still have to reach, there is an adaptation engineering. The same can be applied to reconfiguration in embedded systems.

Focusing on reconfiguration mechanisms instead of on the pursued goal, they have been focused on system level [7, 3] or component level [19, 8]. At system architecture level, components are the smallest working units and the reconfiguration of the system consist of replacing, replicating, updating, ... components. Those mechanism are quite generic and have been externalized to application environments like J2EE. But when delving into a component, few generic mechanisms have been proposed. In [12] a way to introduce self-healing functionalities in java legacy code in a non intrusive way has been defined. [13] has described a framework to dynamically attach a repair engine to a managed application without hardwiring and crosscutting it into the application. The units at with those later works operate are the language constructs: classes, operations, types, ...

978-0-7695-3140-3/08 $25.00 © 2008 IEEE
DOI 10.1109/EASe.2008.11

The works that focus on an architectural level, employ a model of the system: the architecture. But most inside-component reconfigurations are application dependent and many times not distinguishable from the functional part. This increases the software complexity because the reconfiguration mechanism is hardwired with the functional aspects of the component. This paper proposes the usage of models as the basis for component reconfiguration applying it to statecharts in particular. Being able to define reconfiguration plans in the same terms used at component modeling permits the definition of general mechanism suitable to particularize to each application.

The abstraction level at which our work operates resides between the system architecture architecture and implementation languages's constructs. There are many applications domains where statecharts or other formalizations like Petri nets or GRAFCETs could drive the specification, design and implementation of each component. There are many software patterns and implementations aiming to aid the software development in case state machines are used. Statecharts have also gained much attention in the context of Model Driven Engineering (MDE). For example [18] has defined a programming style for the efficient implementation of statechart for embedded systems. Commercial tools exist that map statechart models to code skeletons. More recently, W3C has published the SCXML working draft [6] that consists of a JAVA engine capable of executing statechart from a XML specification. Generally, the focus of attention is on driving and facilitating component development.

We will present a framework called FraC for the creation of software components based on statecharts. In addition to supporting a MDE style of development, it creates a reflective architecture of the software component without any involvement of the developer. This reflectiveness gives to software components the capability to modify their model at run-time. We will take advantage of this reconfiguration capability for various purposes like self healing and fault tolerance. Section 2 will list some reasons and motivations for component reconfiguration and will show our focus of attention on each of them. Section 3 deals with some state machine implementations revealing their lack of support at run-time and will present out framework FraC, a library for statechart based component development in a MDE style of development which adds some run-time capabilities to software components.

2 Self-Healing, Adaptation or Fault Tolerance

Self-healing can be defined as the ability to discover the system's malfunctions and find an alternate way (for example reconfiguring the system) to keep the system working. But its scope is not well defined yet [15]. Some authors see it as a subset of traditional fault tolerance and others as a complementary discipline. There is not disagreement in that self-healing systems must have an error detecting and problem diagnosis capability. In order to add self-healing and adaptation capabilities to a software component, it is necessary to define which symptoms we are going to respond to and how to monitor them.

The causes of system malfunctions could be very different: processing hardware errors, the deployment of a new version of some application, environment anomalies, sensor and actuators errors, ... Depending on each particular system different considerations are taken. For example, in enterprise information systems hardware fault tolerance can be necessary because of availability reasons but if we move to automatic manufacturing plants' control systems, hardware crashes occur much more frequently in sensors and actuator than in processing nodes. Also environmental conditions may affect the sensors' proper working. As mentioned in [15], inexact requisites or lack of them can be the root cause of system malfunctioning and thus, if the software has some capability to cope with this situation, it also could be considered a self-healing ability. Thus, self-healing being so broad, we have to delimit our area of actuation. In our case we will not consider processing node's failures but only environment uncertainties or failures and software defects.

The self-healing ability is an utopia toward which we are moving. Self-* systems must have a control loop and to implement that, it is mandatory to have what we are going to monitor and how we are going to actuate on the system well defined. In a first stage, human intervention is needed to implement this control loop but it is hoped that in the future systems will gain more autonomy. The traditional software maintenance process can be considered the very first control loop: we deploy a system and after detecting some problems, we fix them, then redesign and reimplement the system and finally we deploy a new version. When we foresee some kinds of failures, we can design a system to cope with them but in a heterogeneous systems and when creating a system composing software components, failures derived from the composition itself can arise that require the refactoring of some components of the system. And what is more important, not always are there clear the sources of faults nor what to monitor. We learn about them through the observation of the system's operation. If we could speed up this process, it would help in a human learning process that would eventually automatize the adopted actions to heal the system. Therefore, making the system able to self-heal.

Self-healing involves some dynamic ability and according to a system's dynamic capabilities for self-healing, we can classify a system in one of the next groups: systems with no dynamic ability, systems with application and problem specific capabilities, systems with generic run-time reconfiguration capacities but static strategies for self-healing, and finally, the systems whose healing strategies also can vary at run-time. Our work is in the third group's area, giving software components run-time reconfiguration capabilities and giving the final system developer or integrator the possibility to define concrete healing plans according to each system. We are trying to define reconfiguration mechanisms that are orthogonal to the functional aspects but that works at model level.

There is an overlapping area among self-healing, adaptation, fault-tolerance, ... But reconfiguration can constitute a basis layer upon which those properties can be constructed. For now on, we will focus on a reconfiguration mechanism for a components' model. The framework that we will present in the next section is applicable for software components suitable to be modeled by means of statecharts. This framework will offer some mechanisms to monitor and reconfigure these software components in statecharts' terms. Not in an application dependent way. These will be constructed based on the formers.

3 Statechart Implementation and the Framework *"FraC"*

When implementing statecharts multiple criteria can be applied: memory consumption, execution overhead, development simplicity, extensibility, etc. Even prior to hierarchical state-machines (statecharts), lots of work had been done on state machine implementations. The history begins with two nested `switch` statements, where depending on the actual state first and the received event second, the action to be executed is chosen. With a *state table* having a function for every [event, state] pair, the event dispatching is externalized a bit. With the advent of object orientation, appeared the famous State pattern and many others like the Three Level FSM whose intent was to facilitate the development and extension of state machines.

With statecharts (hierarchical state machines, [14]), being a formalism to facilitate the modeling of state machines, implementation becomes more complicated. With the adoption of statecharts as part of the UML formalism and the wave of MDA, a synergy has occurred among them while trying to facilitate statechart development. The designer need only work at model level, fill some well defined particularizations and automatically obtains the statechart implementation. Any complex decision is delegated to a CASE tool (like IAR Visual State) or a framework. [18] has defined an optimal statechart implementation for C/C++ that can be seen as a collection of programming idioms, there is a little deviation from UML standard, though. [16] has filled those gaps but also has the idea of automatic code generation in mind. In [1] an extensive list of design patterns for finite state machine implementation is presented.

The common factor among most implementations is their intent to facilitate software development. They provide facilities to separate behavior from logic and context interface at development time, but the product code could result in a mess of them making the model get lost at runtime.

One of the patterns that has some run-time ability is the *"Reflective State"* pattern [11]. For example, in case of a software component controlling some hardware element, it helps modify the state machine behavior at run-time to reflect mode changes. For example from *"correct"* to *"abnormal"* working. Even if it helps to reduce software complexity, this run-time capability has to have been contemplated at design time.

3.1 The FraC Framework

In this section we are going to describe the FraC framework which is oriented to the development of statechart based components written in JAVA. Those components communicate through asynchronous messages with reception receipt in order to maintain a low coupling. The connectors among them have been implemented as message queues and the messages are transmitted using CORBA but any other middleware could be used.

As it can be seen in Listing 1, state machines are created inheriting from `StateMachine` and filling the shown methods. `initStructure` serves to create the hierarchical state structure and `initBehaviour` defines the behavior that the statechart must have in response to defined events.

```
public abstract class StateMachine
{
  ........
  protected abstract void initStructure();
  protected abstract void initBehaviour();
}
```

Listing 1. Statechart base.

First of all, following the same philosophy of [5], it permits a Model Driven style of development. To illustrate this, consider the statechart of Figure 1. Listings 2 and 3 show some code snippets of its implementation.

```
s0=new XorState();
s1=new XorState();
s0.addState(s1);
s2=new AndState();
s0.addInitialState(s2);
r1=new Region();
s2.addRegion(r1);
s21=new XorState();
r1.addInitialState(s21);
```

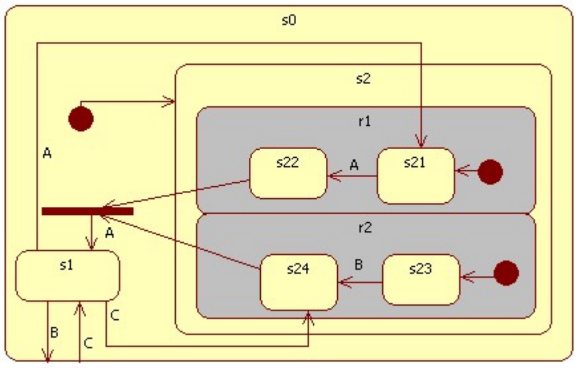

Figure 1. An statechart example.

```
........
```

Listing 2. State structure definition.

```
s24.addReaction(EvC.class,
new SimpleReaction(s23,null,"evC24_23"));
s2.addReaction(EvC.class,
new SimpleReaction(s1,null,"evC2_1"));
s24.setExitAction("exit_24");
s0.setEntryAction("entry_0");
s1.setEntryAction("entry_1");
.......
```

Listing 3. Statechart behaviour definition.

Even if the developer was unaware of its internals, the framework implicitly creates a reflective architecture of the component. The component we employ that communicates asynchronously and the main part of a software component created with FraC are illustrated in Figure 2.

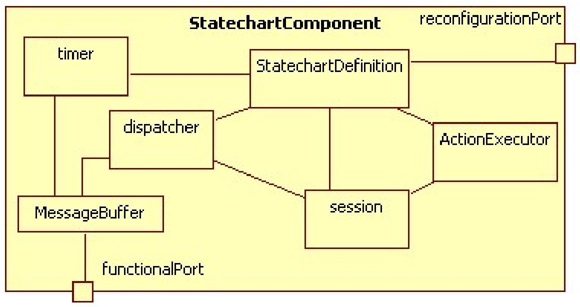

Figure 2. The component structure with FraC.

- `Timer` is an element that serves as a facility for time-out events.
- `Session` is a component's global repository that stores the current active state and the message being processed.
- `MessageBuffer` is where incoming messages are deposited through `FunctionalPort`.
- `Dispatcher` is the element that once the statechart gets stationary, picks the next message and interrogating the `StatechartDefinition` part, instructs the `ActionExecutor` to carry out the corresponding actions in a precise order. In the interrogation phase `ActionExecutor` is also required to evaluate guards.
- `ReconfigurationPort` is an element by which the component can be externally instructed to make changes in the model.

The two main parts of our components are `StatechartDefinition` and `ActionExecutor`. The first is an assembly of objects that reflects the statechart model. It constitutes a meta-level of the component. The second is the one that evaluates the guards and executes the pertinent actions. The `StatechartDefinition` part has the information about the structure and behavior but the dispatcher is the one that has all the knowledge of how to fire actions as a reaction to incoming messages. This division permits us to adjust the statechart model at run-time modifying the `StatechartDefinition`, provided that the `ActionExecutor` has enough capability of guard evaluation and action execution.

Figure 3 illustrates in a simplified manner the interaction among different parts in response to the reception of a message. The `:State` and `:Reaction` multiobjets are elements of the `StatechartDefinition` part.

Comparing this structure with others like the one proposed in the *Reflective State* pattern [10, 11], in this second one the states' internal behavior is what can be changed but not the overall state machine behavior. For that, it is necessary to be able to add/delete states, transitions, etc. In short to change the model. There is another point that makes other patterns more restrictive, most of them have hardwired the logic for transition making inside states, making it impossible to change the statechart model.

3.2 Model Changing

First of all, let us say what we mean by model changing. In section 3.1 we have seen how the `StatechartDefinition` holds the model and the `Dispatcher` interrogates it in order to instruct the `ActionExecutor` to evaluate the pertinent guards and execute the pertinent actions. The `StatechartDefinition` is an object structure that

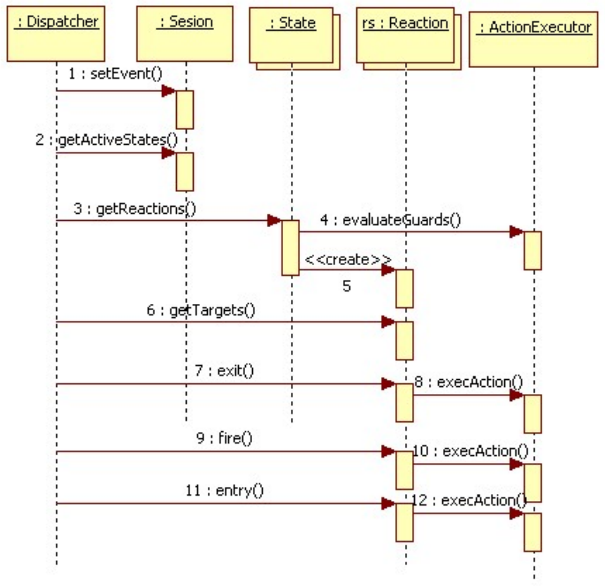

Figure 3. Event Processing

reflects the statechart model. Thus, within our framework, changing the model means changing this object structure.

To illustrate this with a small example let us imagine that we have a distributed system to control the temperature of the rooms of an entire building. Each room has a local controller that when receives an order from an scheduler is responsible for monitoring the assigned room and actuating in the heating/cooling devices. We have an element whose design is shown in Figure 4 in a simplified way.

Let us assume now that we want to extend this component's behavior to prevent the heater or cooler from working when the windows are opened by detecting those abnormal situations as an excessive permanence in the heating or cooling state as partially illustrated by Fig. 5. The immediate solution is to develop a new version and replace the older. But now, we wonder if a more agile way of developing a new version of the component exits. This new version only requires to change the statechart model part. Let us see in the next example how can we obtain this without redesigning the component.

Listing 4 shows how the extension illustrated in Figure 5 can be done without accessing the source code by means of inheritance. In this case we use the base component as a white box. However, realize that the reflection of the statechart model is created in the constructor of the class. Therefore, at initialization time. That means that the component has the ability to change the model at run-time. This ability can be the basis for a more structured run-time model change. The real benefit is that we can change the run-time

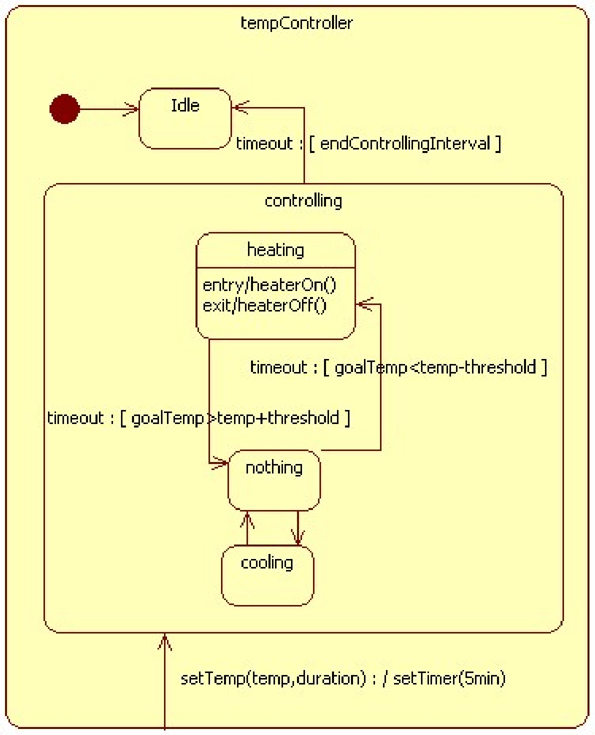
tempController
Idle
timeout : [endControllingInterval]
controlling
heating
entry/heaterOn()
exit/heaterOff()
timeout : [goalTemp<temp-threshold]
timeout : [goalTemp>temp+threshold]
nothing
cooling
setTemp(temp,duration) : / setTimer(5min)

Figure 4. Room temperature controller.

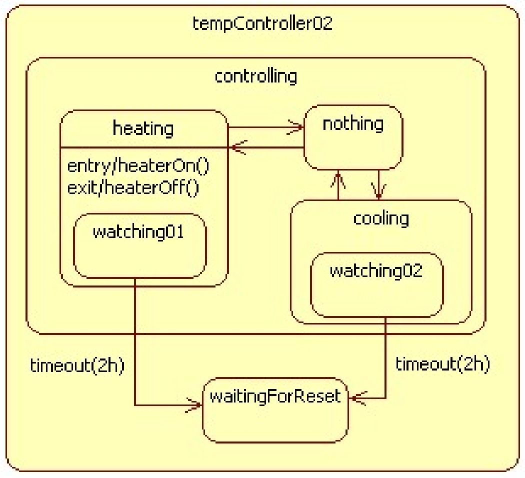

Figure 5. Extended Room Temperature Controller.

model, that can be seen as a mode change, without envisaging those modes at design time. The infrastructure that

FraC creates allows it.

```
class InheritedSM extends BaseSM
{
  protected XorState sWatch01,
              sWatch02,sWaitReset;
  public void initStructure()
  {
    super.initStructure();
    sWatch01=new XorState("watching01");
    sWatch01.setTimeout(2*60*60);
    sHeating.addState(sWatch01);
    sWatch02=new XorState("watching02");
    sWatch02.setTimeout(2*60*60);
    sCooling.addState(sWatch02);
    sWaitReset=new XorState("waitingForReset");
  }
  public void initBehavior()
  {
    super.initBehavior();
    sWatch01.addReaction(EvTimeout.class,
          new SimpleReaction(sWaitReset);
    sWatch01.addReaction(EvTimeout.class,
          new SimpleReaction(sWaitReset);
    sWaitReset.addReaction(EvReset.class,
          new SimpleReaction(sIdle);
  }
}
```

Listing 4. Code for the statechart extension of Figure 5 in order to fulfill new requirements.

3.3 Composition

When creating a system by assembling components, many problems can arise: platform incompatibility, required/provided interface mismatch, semantic divergence,... Apart from the verification of each component by itself, the composition itself must be verified and some components must be adapted to fulfill system requirements. This adaptation can be done with centralized controllers, wrappers or other kinds of glue code, or each component can be programmed to adjust itself to each particular system it is going to be part of. The key problem is that the developer of the component can not foresee all the situations in which the component will participate.

Our framework offers a complementary alternative in doing such an adaptation as demonstrated in [4]. Let us continue with the previous example and imagine that we specify a new requirement for the room temperature controlling system. Suppose that we put a presence detector in each room and we want the temperature controller to stop if the presence detector has been detecting nobody for more than 5 minutes. The initial presence detector working is shown in Figure 6.

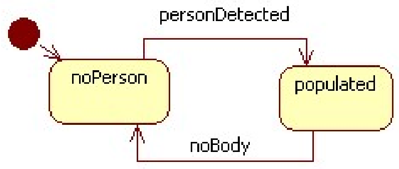

Figure 6. Presence detector statechart.

We can formulate this new specification saying that the room controller must go to `idle` state if the presence detector has been more than 5 minutes in `noPerson` state. Let us see a solution which does not need to refactor each component nor insert glue code, but upgrades each component changing their behavioral model. As can be seen in Figure 7, we have enhanced the initial room controller with a new transition from `working` to `idle` upon the reception of event `stop`(Figure 8). And the presence detector has been upgraded with two substates so that it can notify the room controller the event of being detecting no person for more than 5 minutes. Remember that our framework has a timeout option that sends a timeout event after some time from the entry of the associated state. Even if this example is very simple and incomplete, it illustrates an alternative to resolve some composition issues exclusively working at model level. The facilities that FraC offers can be seen as composition primitives.

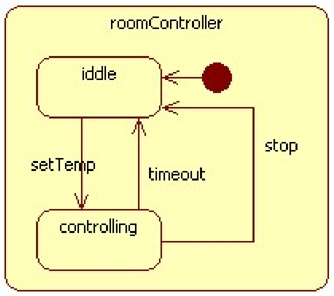

Figure 7. Upgraded room controller behavior.

3.4 Software Fault-Tolerance

N-versioning and Recovery-Block approaches are general mechanisms to cope with software failures that need to be particularized for each application. For example the granularity of the module to be replicated or that can have a failure must be determined . Due to this the next doubt

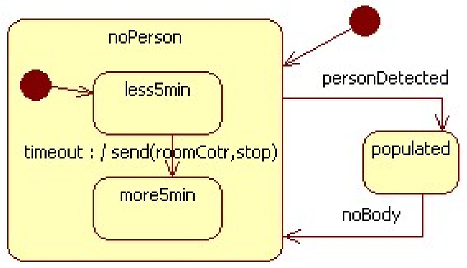

Figure 8. Upgraded presence detector behavior.

arises: if we knew that a component followed the structure above mentioned, could this serve in the determination of those modules.

In [9] there is an extensive field study and classification of software faults. In this study the assignment, checking and algorithm defects are around 82%. These kinds of errors are the ones that can be made in the guards and actions of a statechart. In the presented statechart structure all these are gathered in the action executor part. Thus, it is enough to change or replicate this executor as a support for software fault-tolerance.

In the executor part we have put an interceptor as illustrated in Figure 9. This extra indirection is to catch the exceptions that are thrown when executing an action or evaluating a guard. In response to those situations, Figure 10 shows how our framework implements a recovery-block strategy changing the actual executor part at run-time. When the executor handler receives a command from the dispatcher, it redirects it to the responsible executor. If this one can not process it, the executor handler knows it and decides to redirect the command to another executor if this redundancy has been procured. Of course, this mechanism must be enhanced with some other patterns depending on the particular application. For example, the *"pseudostate"* variables must be restored to a value prior to the first command processing. *Memento* pattern can be used for this issue.

The most general event processing that a statechart can have to execute is when it has to make a transition: it has to evaluate some guards, it must exit some states, then the actions associated with the transition must be executed and eventually, it must entry the target state and possibly, some of its superstates and substates. The granularity for fault detection the framework offers is the one offered by each of those individual actions. For example, if we catch an exception when exiting a particular state, the executor handler will replace the executor, retry the failed action and follow

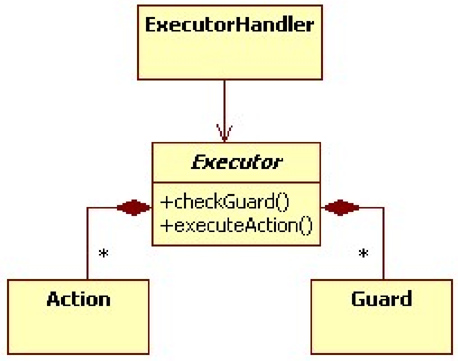

Figure 9. Executor Handler governs the executor part.

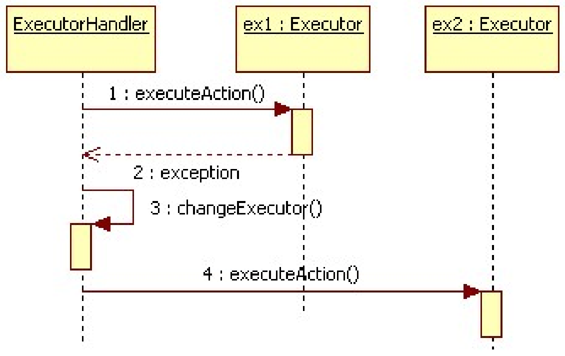

Figure 10. Executor part change in response to a exception.

executing the previously mentioned sequence.

Finally, some assumptions must be made about the possibilities of faults in the statechart definition or the control part. In this experimental implementation the control structure of the component is defined using a framework and potentially helped by an automatic code generation tool. This framework is supposed to be tested much more than any particular application that uses it. Thus, the possibilities of software defects in it are far fewer. The possibility to have a bug in a particular control definition is more probable from incorrect specification than from software fault. For that, we will belittle the possible fault in control related to the execution part witch is more sensible to implementation bugs.

3.5 Self-Healing

Having described the run-time capabilities that the FraC framework provides to statechart based software components, let us explain our formula to give self-healing abilities to software systems. From our point of view not only self-healing but any other run-time ability needs to be supported by reconfiguration mechanisms. When an abnormal situation is detected the system must fire a reconfiguration action to deal with it. As we have said in section 2, our work is directed toward defining a generic reconfiguration mechanism that operates at model level upon which we can construct application specific healing plans. We have defined a reconfiguration mechanism that is orthogonal to the functional aspects but that works at model level.

We want to materialize and particularize for statecharts the mechanism described in [19] for adding self-healing capability to a system. This mechanism is distributed in each of the system's components. It monitors each of them and upon detecting an abnormal situation, it isolates the component be means of the connector it uses and restores the component. In the meanwhile, the healing layer of one component can notify other components to such situation because some local problems has to be accomplished globally.

The `reconfigurationPort` that appears in Figure 2 serves for the same purpose. At the moment it has the capability to modify the model at run-time. Those modifications are the same as the ones that we have previously shown using inheritance: add states, transitions, generate new timeout events, ... This group of facilities by itself is not really useful when accomplishing a system wide reconfiguration at run-time or even for a unique component reconfiguration. They serve as a basic layer upon which the final system integrator can construct particular healing actions.

Those healing actions are application dependent and in consequence, very variable. In some cases we need to change the interaction among multiple components. In other cases, we must change some components' statechart model in order to change its behavior mode. Each component can be supplied with many and related statechart models, each of those corresponding to a different working mode. Upon the detection of an abnormal situation the component can fire an especial event that instructs the component to swap to another statechart and if necessary, it can notify other components through their reconfiguration port and cause in other components a mode (model) change. When a software failure is detected, in addition to procuring redundancy, we can fire some of the previous actions. Those capabilities that FraC gives to software components have been described throughout section 3.

Let us say that many times the self-healing term is not clearly enough distinguished form other terms like adaptation. In [17] an environmental fault is defined as an error in the state of the environment. In the previous example, it could be that the thermometer does not work properly. In aggressive environments like industrial ones, control systems have to deal with many exceptional situations mostly caused by sensors and actuator failures and environmental uncertain conditions. For this kind of situations FraC offers a support for easier adapt the system by means of reconfigurating some of its components.

4. Conclusions and Future Work

Model driven engineering has advanced during the last years but it main focus has been the improvement of software development. Few initiatives aim at maintaining models till run-time. Reflective architectures provides mechanisms for changing structure and behavior of software systems dynamically. This paper is positioned in model driven engineering context aiming at transforming the model in a reflective architecture.

This reflective architecture permit us to make reconfiguration actions in the same terms used when modeling software components. Even if it is not used at run-time, it makes easier to extend or modify existing components if those modifications can be expressed as model variations. Furthermore, the developer of the original component need not to contemplate the possibilities of such modifications. The used framework, provides this ability in an implicit way.

It is not clear how advantage of the presented run-time capability can be taken. It can constitute a generic basic layer in which reconfiguration strategies particularized for each concrete application can be defined. Its main benefit would be that the original software components' designer need not consider any reconfiguration mechanism at design time. The FraC framework offers a basic language by which the integrator of the components, the one will build the final system, can define concrete reconfiguration plans.

References

[1] P. Adamczyk. The anthology of the finite state machine design patterns. *The 10th Conference on Pattern Languages of Programs*, 2003.

[2] R. Adler, D. Schneider, and M. Trapp. Development of s&r embedded systems using dynamic adaptation. In *M-ADAPT, 1st Workshop on Model Driven Software Adaptation*, 2007.

[3] J. Apavoo, K. Hui, C. Soules, R. Wisniewski, D. D. Silva, O. Krieger, M. Auslander, D. J. Edelsohn, B. Gamsa, G. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1), 2003.

[4] X. Aretxandieta, X. Elkorobarrutia, and F. Barbier. Component adaptation for correctness in composite systems. To be published in *7th IEEE International Conference on Composition-based Software Systems, ICCBSS*, Feb 2008.

[5] F. Barbier. Mde-based design and implementation of autonomic software components. *International Conference on Cognitive Informatics (ICCI)*, 2006.

[6] W. Consortium. State chart xml (scxml): State machine notation for control abstraction. working draft. *http://commons.apache.org/scxml/*, February 2007.

[7] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self healing systems. *Proceedings of the first workshop on Self-healing systems*, pages 21–26, 2002.

[8] P. A. de C. Guerra and C. M. F. Rubira. An idealized software fault-tolerant architecture component. In *Workshop on Architecting Dependable Systems, ICSE'02 International Conference on Software Engineering*, 2002.

[9] J. A. Duraes and H. S. Madeira. Emulation of softwatre faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32:849–867, Nov 2006.

[10] L. L. Ferreira and C. M. Rubira. Reflective design patterns to implement fault tolerance. *Workshop on Reflective Programming in C++ and Java, OOPSLA98*, 1998.

[11] L. L. Ferreira and C. M. Rubira. The reflective state pattern. *Pattern Languages of Programs, PLoP'98*, 1998.

[12] M. M. Fuad, D. Deb, and M. J. Oudshoorn. Adding self-healing capabilities into legacy object oriented applications. *2006 International Conference on Autonomic and Autonomous Systems ICAS '06*, jul 2006.

[13] R. Griffith and G. Kaiser. Manipulating managed execution runtimes to support self-healing systems. *Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005)*, 2005.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[15] P. Koopman. Elements of the self-healing problem space. *Workshop on Software Architectures for Dependable Systems(WADS2003)*, 2003.

[16] G. Pinter and I. Mazjik. Program code generation based on uml statecharts models. *Periodica Politechnica*, 47(3-4):187–204, 2003.

[17] C. M. Rubira. Structuring fault tolerant object oriented systems using inheritance and delegation. *PhD Thesis, Dept of Computing Science, University of Newcastle upon Tyne*, Oct 1994.

[18] M. Samek. *Practical Statecharts in C/C++: An Introduction to Quantum Programming*. CMP Books, 2002.

[19] M. E. Shin. Self-healing components in robust software architecture for concurrent and distributed systems. *Science of Computer Programming*, 57(1):27–44, jul 2005.

[20] K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer. A system model for dynamically reconfigurable software. *IBM Systems Journal*, 42(1):45 – 59, Jan 2003.