A Self-Healing Mechanism for State Machine Based Components

Xabier Elkorobarrutia, Alberto Izagirre, and Goiuria Sagardui

Xabier Elkorobarrutia, Alberto Izagirre, Goiuria Sagardui
Departamento de Informtica
Mondragon Goi Eskola politeknikoa
20500 Mondragn (SPAIN)
Tel.: +34 943 794700
xelkorobarrutia,aizagirre,gsagardui@eps.mondragon.edu

Abstract. This article describes a self-healing mechanism for state-machine based distributed components. Each component is composed of two layers: a healing (HL) and a service or functional layer (FL). At least, the functional layer must be implemented according to a state-machine specification.
The healing layer has the capacity of monitoring the service layer of the component of witch it is a part and responsible of. In the event of a failure of the functional layer, the healing layer acts on it in terms of states. It will be shown that this mechanism, being consistent with classical fault-tolerance strategies, allows the HL to act on the FL in a more precise way than employing an entire software module as a replacement unit. It will be shown also that this mechanism is suitable for hot-swapping software modules.

1 Introduction

Autonomic Computing (AC) in general, and Self-Healing in particular, have gained much attention but it's not very well defined in terms of scope [3, 10]. The are many works oriented to highly distributed and heterogeneous corporate systems, where the main obstacle to further progress in the Information Technology (IT) industry is our inability to manage the system as a whole, instead of focusing on individual software environments. In [4] a roadmap to accomplish this problem is proposed and the core of the proposal is to make the software self-aware and give it the ability to self-manage, i.e. Autonomic Computing.

In those works, the characteristics of a autonomic systems have been further classified in self-configuring, self-optimizing, self-healing and self-protecting. The necessary ability to accomplish them is to be able to monitor the system. Each of these topics has had its focus of attention: in [6] is proposed an Interceptor based approach to detect some constraint violation and generate these interceptor automatically from XML specification. In [5] it is shown that hot-swapping could be an enabling technique for AC, being interposition and replacement of

code, the way to accomplish it. In [11] is introduced a relationship among autonomic elements and multiagents systems, and based on agent technology, they propose an architecture for enabling autonomic computing. In [12] an architecture changing healing mechanism is proposed; for that, they work on event based systems where components interacts via connectors.

On the other hand, instead of proposing different techniques or mechanisms that could provide a self healing ability to a system, Shin [1] has proposed a self healing mechanism for components based on a design method described by Gomaa [2], enhancing this design method. This will be the base for our work. In section 3 we express our point of view of what self-healing could mean trying to differentiate from fault tolerance.

In section 2 we state our positioning with respect to state-machines and MDA, and also we describe our working context. In section 3, we try to clarify a little bit the term self-healing. Section 4 and 5 describes a mechanism that can be used to heal a component and other uses or possibilities of it, respectively. And finally, section 6 concludes this paper.

2 State Machine Based Components

State machines allow precise specification of the behaviour of a structural element. But even more, it is a key abstraction in the context of Model Driven Architecture (MDA) because of its ability to express the behaviour of reactive systems. The key elements of MDA are *model executability* and *model transformation*. With the former we aim to verify the system at the model level; and the latter aims to constraint the code by the model instead of being the model only a blueprint. Fig. 1 summarizes the core of MDA [8].

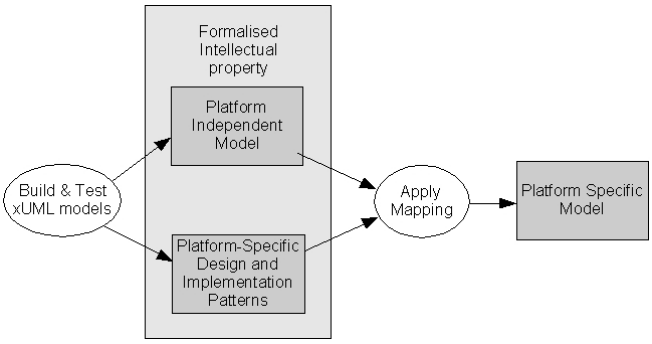

Fig. 1. Model Driven Engineering.

We are using state machines as a mechanism to specify the inside of components. But beyond that, the possibility to operate at run-time on the component

in terms of states, allows one to develop a management code that could reconfigure or repair the component. Of course, this facility would not be known from the functional perspective; it only has management purposes. In fact, we are proposing *non intelligent* autonomic software components that have enough prefabricated capabilities to support self-configuring and self-healing activities.

At this moment, we are working with a framework called *PauWare* [13] developed by the University of Pau. It is an engine for executing statecharts in a JAVA environment. At the present time it has two basic reconfiguration options that consist of forcing a component's state machine to a determined state and returning to the prior state the component was on.

Obviously, it has no sense to invoke these possibilities from the state-machine itself. We need to have another entity besides the state machine that monitors it and in the event of failure or other circumstance of interest, acts on it. Those other circumstances involve all the assembly of components from witch any action must be deduced. This other entity besides the state machine we have proposed, must in many cases act from a global perspective, it needs to be a distributed element across all the components and the part that resides in each of the component is what we have called healing-layer.

3 Self-Healing vs Reliability

As Koopman [10] states, there is not a clear distinction between SH and reliability. e.g. does SH enclose fault tolerance? Or is it vice versa? Or do they have an overlapping area? Instead of classifying our work within one or another term, we will say that we aim to heal a component from software and environmental errors, being these last ones due to communication or neighbour components.

Looking first at fault tolerance, in works like [7] we can see that a common denominator for fault tolerance is replication. Replication is the way we obtain fault-containment regions. We replicate nodes, servers, applications depending of the failure modes that have been contemplated. Putting them working together, we pretend that, if the fault assumptions are satisfied, the element in consideration never fails. That could be perhaps the element that differentiates fault tolerance from self-healing. In the former we try to obtain perfect components; in the second, we assume that in some circumstances, some element will have to be repaired. Obviously, a replica also will have to be repaired in case of failure; but, from the functional point of view, it's the replica that is recuperated and not the component .

Going deeper into software failure recovery, block-recovery and N-version programming are two classical strategies. In the former, in the presence of a software failure, we retry to give the asked service by another functionally equivalent module or software block. In the latter we put executing in parallel functionally equivalent modules and a voter chooses the supposed correct response. In both cases, the granularity of the *software module* is open; but it has the connotation of a service whose bad response could be detected by the value of the response or the delay of it. In very close relation with block-recovery strategy there is

the exception mechanism, supported by various programming languages. Every time a software block raises an exception, we can try to *heal* the context (if the cause of the failure is in it) and retry the same or equivalent software block. In all cases, we are putting all replicas in the same state of departure.

Based on this last fact, we will try to extend the recovery points no only to some initial known states but potentially, to every state in the state machine based on the capabilities mentioned in section 2.

4 State Based Self Healing Mechanism

For the sake of discussion, we will use the simple state machine depicted at Fig. 2 that does not use nested states nor parallel regions. When an event of interest occurs, a transition fires and at the most complete scenario, the actions associated with the `exit` section of the present state, those associated with the transition itself, and finally, the actions of the `entry` section of the new state are executed. But what if we detect that an exception has occurred on one of the actions?

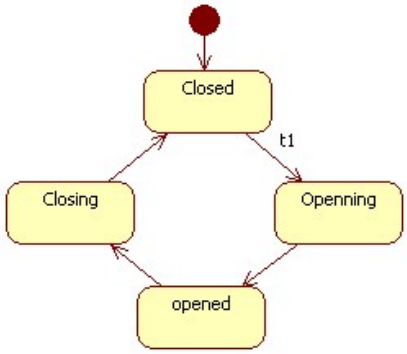

Fig. 2. A simple state machine.

Suppose that we have another version of the same state machine specification available at run time; lets call them $V1$ and $V2$. If the transition $t1$ fires, bringing the state machine from state *Closed* to *Opening*, we could make an equivalent transition form state *Closed* of $V1$ to state *Opening* of $V2$. Having the capability at run-time to operate on the state-chart implementation we are migrating a statechart from one implementation to another as illustrated by Fig. 3 were $t1$ and $t1SH$ are supposed to be identical except the reaching state.

The above mentioned mechanism is similar of what happens in the CORBA object model. A CORBA object is a virtual entity that is accessible by means of an Object Request Broker (ORB). At run-time, an implementation mechanism

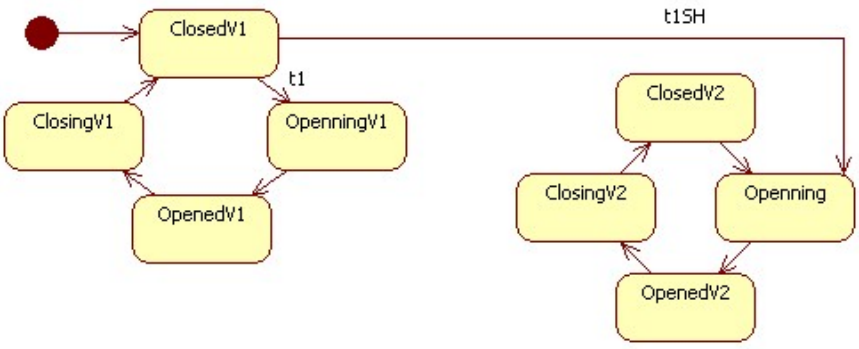

Fig. 3. State-chart migration.

called a servant (e.g. a class, if we are using an object oriented language) incarnates that CORBA object. This CORBA object could be etherealized, meaning that there is not a servant that can give the service the object is supposed to support, but it could be incarnated another time in other servant and become available. In other words, we have a unique virtual CORBA object that could migrate from one servant to other. What we have proposed by the example before, is analogous: we have a virtual statechart that is the specification of a component that could migrate from on implementation to another.

This mechanism for migration could resemble a version changing through hot swapping. But let focus at self-healing. If we put together with the statechart a healing layer with the possibility to monitor it and has the capacity to redirect a transition from a state of implementation $V1$ to a equivalent state of $V2$, then we could perform some kind of self-healing (see Fig. 4).

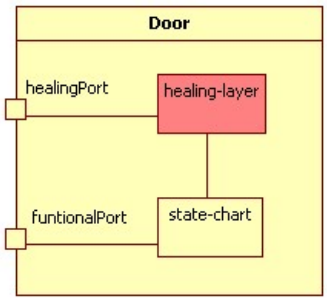

Fig. 4. An example componet layers.

Each event that fires a transition, triggers the actions associated with the entry section of the leaving state, the fired transition itself, and the reaching state. If the event does not fire a transition, fewer actions groups are involved. As an example, supposing that the execution of the statechart may throw exceptions due to software errors or some missing considerations at implementation time, the healing layer could catch them and fire the above mentioned migration mechanism. It will be also possible if we insert code that verify some set of invariants to be meet in the entry of each state.

For that, it could be necessary to do some cleanup and, depending on which action set has occurred the failure, employ the facilities of state-forcing and returning to a prior state we mentioned in section 2 (we need to record the last event). In other words, we are hot-swapping between different statechats implementation versions to self-heal the component. It seem to resemble the *Memento* pattern [14] but by definition, a state reflects the past story of a pure state-machine. The exception to this are those internal variables whose different values are not reflected in the state machines (each value means a different state). But in this case, they should be in the specification and we could apply the memento pattern to them. Thus, we add some preparation work in the migration from one implementation to another; but we could also put these variables in a common place where different implementations could reach them.

5 Software Engineering Considerations

But besides this simple self-healing mechanism for self-healing state-machine components, we see other opportunities where we can apply it. The first and the most obvious is version changing. The second, affects all the assembly of components. If a component can not accomplish its responsibilities, it could migrate to a version where a reduced (degraded) set of functionality is provided. This situation could be informed to the rest of the components causing a possible reconfiguration in them.

But if a component is not able to perform his work and this can affect the rest of component in the assembly, it is a situation that has to be contemplated at an initial stage of the design of the system. In this case, we can talk about different modes of functioning of the entire system that actually constitutes a high level global state machine model. Whenever the system has to change its mode, also each of its constituent components has to do so. If the behaviour of a component in different modes is quite different from one to another, putting them together could result in a voluminous model. To facilitate its development, we could develop independently a model for each mode an let the healing-layer to swap across them. Those global functioning modes provide us another criteria for division that could facilitate the development of the system and its complexity.

6 Conclusions

This paper has described an approach of inserting a self-healing mechanism in components that are specified according to a statechart and whose implementations also offer the possibility to act on them in terms of state; i.e. forcing the component to some state and rolling back one transition. It has also been shown, that this mechanism being another face of the block recovery strategy, can also be used as component implementations hot-swapping and version-changing at run-time.

Finally, it has been shown that in case of mode change of the entire system, of which a component is a part, using this mechanism facilitates the division of design and implementation of the component.

At this moment, we are trying to enhance the *PauWare* statechart engine by means of capabilities to self-healing. We must bound the failure types we need to cope with. But at the same time, there have been opened other question, e.g. how useful could be the same mechanism in version changing or support for the design of the component. What we have not yet contemplated are the synchronization problems.

References

1. Shin, Michael E., Cooke, Daniel: Connector-Based self Healing Mechanism for Components of Reliable Systems. In: DEAS, St. Louis, Missouri, USA (2005)
2. Hassan Gomaa. Designing Concurrent, Distributed, and Real-Time Applications with UML. Addisson Wesley (2000).
3. Kephart, Jeffrey O., and Chess, David M. The Vision of Autonomic Computing. Computer, **Vol. 36(1)**, 41–52 (2003)
4. An Architectural Blueprint for Autonomic Computing. IBM (2003)
5. Apavoo, J. et al. Enabling Autonomic Behabior in Systems Software with Hot Swapping. IBM Systems Journal, **Vol. 42(1)** (2003)
6. Wang, Q., Mathur, A. Interceptor based Constraint Violation Detection. In: EASe, Greenbelt, Maryland, USA (2005)
7. Jalote, Pankaj. Fault Tolerance in Distributed Systems. Prentice Hall (1994)
8. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie I. Model Driven Architecture with Executable UML. Canbridge University Press (2004)
9. Samek, M., Practical Statecharts in C/C++. Quantum Programming for Embedded Systems. CMP Books(2002)
10. Philip Koopman. Elements of the Self-Healing Problem Space. Workshop on Software Architectures for Dependable Systems(WADS2003) (2003)
11. Gerald Tesauro, David M. Chess, William E. Walsh, Alla Segal, Jeffrey O. Kephart and Steve R. White. A Multi-Agent Systems Approach to Autonomic Computing. AAMAS'04 (2004)
12. Eric M. Dashofy, Andre van der Hoek and Richard N. Taylor. Towards Architecture-based Self Healing Systems. Proceedings of first Workshop on Self-Healing Systems (2002)
13. Pauware. University of Pau. http://www.pauware.com
14. Erich Gamma, Richard Helm, Ralph Johnson, John Vlisides. Design Patterns. Addisson Wesley (1994).