An Enhanced Composition Model for Conversational *Enterprise JavaBeans*

Franck Barbier

PauWare Research Group – Université de Pau
Av. de l'université, BP 1155, 64013 Pau CEDEX – France
Franck.Barbier@FranckBarbier.com

Abstract. When designing applications with *Enterprise JavaBeans* (EJBs) and more specifically with *Stateful Session Beans*, a major difficulty (or even an impossibility) is being able to properly transform business models and more precisely UML 2 models, into such component types, while including the expression of their mutual compositions. This contradicts with the spirit of the emerging *Model-Driven Architecture* (MDA) software engineering paradigm based on the definition of seamless model transformations. In this scope, this paper proposes and describes an appropriate Java library in order to increase the composition power of EJBs. The proposition includes a support for a broadcast communication mode (assimilated to "horizontal composition" in the paper) which is, *a priori*, incompatible with non reentrance, a key characteristic of EJBs. Besides, "vertical composition" is the counterpart of "horizontal composition". "Vertical composition" enables the consistent hierarchical combination of composite behaviors and compound behaviors, both being specified and implemented by means of UML 2 *State Machine Diagrams*.

1 Introduction

Szyperski *et al.* have claimed for a long time that: "Components are for composition." [1]. In other words, all software components are software parts, although not all software parts are necessarily software components. Furthermore, if components are not specifically designed to have composition potentialities (*i.e.*, composability or compositionality attributes) at assembly time, the risk is high that components will fail to interoperate properly. That is the reason why technological component models exist: *Enterprise JavaBeans* (EJBs), CORBA Component Model (CCM), COM+ or Fractal. In providing a well-bounded accurate development and deployment framework, such component models support composition templates. A resulting advantage is that, by complying to the imposed format[1] of components, composition is easy and straightforward. A disadvantage is the difficulty of transforming business models like UML models for instance (which in essence are free of technical constraints) into technical components. In other words, stereotyped components (*e.g.*,

[1] The term "format" is here preferred to that of "model" since technological components obey to code construction and deployment rules rather than to formal/mathematical specifications.

Entity Beans, *Session Beans* and *Message-Driven Beans*) of a given technological component model (*e.g.*, EJBs) may be considered as moulds. Melting business models down in order to fill these moulds is a strong expectation in the software industry.

In the spirit of MDA [2], model transformation rules have to formalize how a platform-independent model (PIM) is transformed into a platform-specific model (PSM). This theoretical principle may however stumble over incompatible model properties. We have carried out experimentations on this problem with UML 2 *State Machines Diagrams* and *Sequence Diagrams*, and more generally with the global "UML 2 Composition Model" [3]. Software components and compositions modeled by means of UML possess recognized features coming from the intrinsic "semantics" of UML itself. The most well-known characteristics are for instance the "coincident lifetime" of composites and compounds in, what we call below, "vertical composition". For "horizontal composition", which is closely related to component communication, broadcast is the underlying communication mode (a heritage of Harel's Statecharts [4]). In EJBs, the predefined composition mechanisms do not conform to these idealistic properties.

This paper proposes a solution for fitting conversational EJBs, which are *Stateful Session Beans*, to the most important conceptual composition mechanisms of UML 2. This occurs through the construction of a dedicated Java library named *PauWare* which is illustrated in this paper.

That is why Section 2 gives a brief overview of EJBs. Section 3 insists on the problem of non reentrance, which is particular to EJBs and which, *a priori*, precludes the implementation of broadcast in EJBs. Broadcast indeed comes from the executability facilities of UML 2 *State Machines Diagrams* and *Sequence Diagrams* and thus, cannot be ignored. Section 4 shows how this has been solved with *PauWare*: code samples are provided. Section 5 is about "vertical composition": how to compose conversational EJBs, hierarchically, starting from the hypothesis that they own and are governed by a statechart that exists inside themselves. A major challenge amounts to synchronizing the two statecharts of a composite and a compound. To conclude in Section 6, we evoke the link of this work with autonomic computing.

2 *Enterprise JavaBeans*

EJBs [5] constitute a technological component standard. They also represent a highly coercive computing framework as far as the format of an EJB is predefined and strict (Fig. 1). From the code viewpoint, an EJB must have a Java implementation class and appropriate interfaces for its clients. From the deployment viewpoint now, an EJB must also have values assigned to mandatory deployment parameters.

Since EJBs' shapes cannot be ordinary and have to satisfy many constraints, EJBs are by their very nature composable. Components that do not comply to standards can indeed be composable with much difficulty. However, in practice, the EJBs' composition model may demonstrate numerous limitations. This is especially the case for conversational EJBs. This specific EJB type offers interesting facilities to programmers. For instance, programmers can control creation decisions; and conversational EJBs remain unshared between clients. Unfortunately, even though it is possible to scrupulously control states within the inside of *Stateful Session Beans*, sophisticated combination of such conversational EJBs is poor. In UML, models such as different state machines[2] may be assigned to distinct business components. Composing these components amounts to taking into account scenarios embodying the communication between them. State machines and scenarios however rely on a composition semantics that has no direct mapping in EJBs. As a result, the benefits from having a formal semantics for statechart composition (see for instance [6] or [7]) cannot really be exploited at the implementation level.

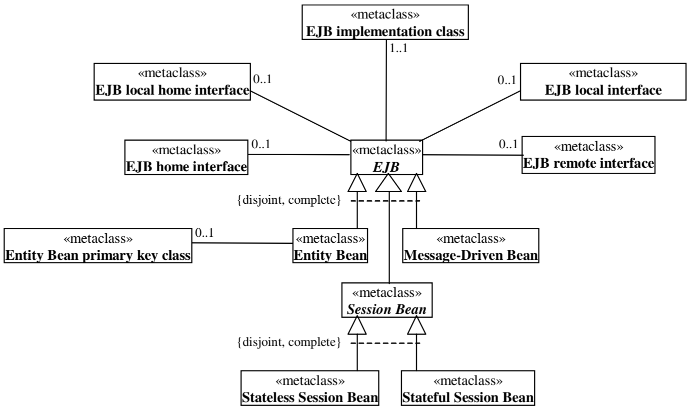
«metaclass» EJB implementation class
1..1
«metaclass» EJB local home interface
0..1
«metaclass» EJB local interface
0..1
«metaclass» EJB home interface
0..1
«metaclass» EJB
0..1
«metaclass» EJB remote interface
{disjoint, complete}
«metaclass» Entity Bean primary key class
0..1
«metaclass» Entity Bean
«metaclass» Message-Driven Bean
«metaclass» Session Bean
{disjoint, complete}
«metaclass» Stateless Session Bean
«metaclass» Stateful Session Bean

Fig. 1. UML metamodel expressing the contractual format of an EJB and the possible types of EJBs

3 Non Reentrance

In attempting to construct the inside of a *Stateful Session Bean* by means of a state machine, one problem is caused by the broadcast communication mode, which is the basis of Harel's Statecharts [4, p. 269]: "The statechart communication mechanism, on the other hand, is based on broadcast, whereby the sender proceeds even if nobody is listening." EJBs do not accept requests while transactions are in progress (this phenomenon is known as non reentrance) while broadcast supposes that requests may arrive at any time.

As an illustration, we reuse the *Railcar control system* case study presented in [8]. In Fig. 2, a railcar that is less than 80 meters far from a terminal, sends *crossing request* which is part of the remote interface of the *Terminal* component type (see the right hand side of Fig. 2). In Fig. 3, the sending of *crossing request* may be observed within the statechart of the *Railcar* component type by means of this expression: ^*my next possible stop.crossing request(self)*. The reception of and response to *crossing request* appears within the statechart of the *Terminal* component type (Fig. 3).

[2] UML 2 state machines are closely related to Harel's Statecharts. We comment on some key differences in the rest of the paper but we do not formally distinguish the expression "state machine" from the word "statechart" all along the paper.

In EJBs, the call of *crossing request* occurs within a transaction that started when *alert80* arrived. The receiver terminal may reply to the sender railcar (by using *self* which is a parameter of *crossing request*) that some passengers standing at the said terminal want to get onto the approaching railcar. How then may one guarantee that *candidate passengers* (the possible reply) is not received at an unsuitable moment, *i.e.*, if the transaction associated with *alert80* is not yet finished?

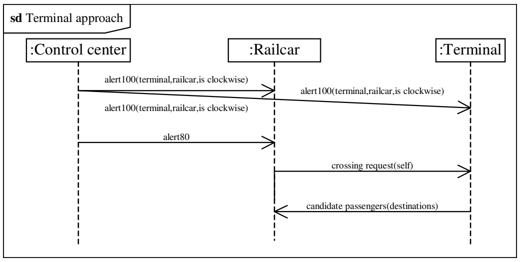
sd Terminal approach
:Control center
:Railcar
:Terminal
alert100(terminal,railcar,is clockwise)
alert100(terminal,railcar,is clockwise)
alert100(terminal,railcar,is clockwise)
alert80
crossing request(self)
candidate passengers(destinations)

Fig. 2. Scenario of communication between three respective instances of a Control center, a Railcar and a Terminal components

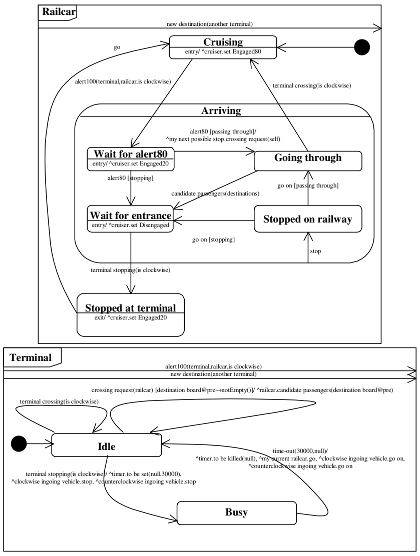
Railcar
new destination(another terminal)
go
Cruising
entry/ ^cruiser.set Engaged80
alert100(terminal,railcar,is clockwise)
terminal crossing(is clockwise)
Arriving
alert80 [passing through]/
^my next possible stop.crossing request(self)
Wait for alert80
entry/ ^cruiser.set Engaged20
Going through
alert80 [stopping]
candidate passengers(destinations)
go on [passing through]
Wait for entrance
entry/ ^cruiser.set Disengaged
Stopped on railway
go on [stopping]
stop
terminal stopping(is clockwise)
Stopped at terminal
exit/ ^cruiser.set Engaged20
Terminal
alert100(terminal,railcar,is clockwise)
new destination(another terminal)
crossing request(railcar) [destination board@pre→notEmpty()]/ ^railcar.candidate passengers(destination board@pre)
terminal crossing(is clockwise)
Idle
time-out(30000,null)/
^timer.to be killed(null), ^my current railcar.go, ^clockwise ingoing vehicle.go on,
^counterclockwise ingoing vehicle.go on
terminal stopping(is clockwise)/ ^timer.to be set(null,30000),
^clockwise ingoing vehicle.stop, ^counterclockwise ingoing vehicle.stop
Busy

Fig. 3. Two communicating statecharts of a Railcar and Terminal components

4 Horizontal Composition of Conversational EJBS

So, the coercive composition model of EJBs precludes intertwined communication, but a consequence of broadcast is that a request receiver may immediately reply to the sender even though the latter is not, from the EJBs' composition model's viewpoint, in an "appropriate" state (while the transaction is in progress).

To solve this problem, we propose a MDA-based Java statechart execution engine that automates the complete and coherent management of statecharts at runtime. The chosen executability semantics is obviously that of UML 2 which is slightly different (even if broadcast remains) from that of the original Statecharts and of some Harel's variants [8]. In [9], two key subtle semantic differences are formally specified: UML 2 advocates a run-to-completion execution model (a first characteristic[3]) which ensures that, within a given state machine instance, the processing of a new request starts when, and only when, the immediately prior request processing is terminated. This mechanism is close to the EJBs' transaction management mechanism. In our approach, requests that may have linked replies, require special treatment so that statechart cycles are not disturbed by impromptu request receptions. Independently of EJBs, this mechanism is for us mandatory in order to keep statecharts consistent throughout execution cycles. A consubstantial result of such an implementation is that the non reentrance constraint imposed by EJBs is automatically satisfied.

From a design viewpoint, this simply leads to incorporating a statechart into the Java implementation class of a *Stateful Session Bean* as follows (code is incomplete):

```
protected Statechart _Arriving = new Statechart("Arriving"); // + the
other states

protected Statechart_monitor _Railcar = new
Statechart_monitor((_Arriving.xor(_Cruising)).xor(_Stopped_at_termina
l),"Railcar");
```

Next, coding *alert80* leads to what follows (code is incomplete):

```
_Railcar.fires(_Wait_for_alert80,_Going_through,passing_through,_my_n
ext_possible_stop.getEJBObject(),"crossing_request",args,Statechart_m
onitor.Broadcast); // + the other transitions

_Railcar.run_to_completion(); // non interruptible statechart cycle
```

In the code above, *crossing request* (in bold print) is called by means of the Java reflection API. The *Statechart_monitor.Broadcast* parameter value must be used if the specification shows that a reply to the sent request (*i.e.*, *crossing request*) is probable. Since this mechanism is costly, it has not been generalized within *PauWare*. Programmers have thus to pay attention to possible faults caused by reentrance.

5 Vertical Composition

The need for rich composition not only obliges one to have "horizontal" composition, but also "vertical" (a.k.a. "hierarchical") composition. As an illustration, the *Fractal* composition model [10] supports hierarchical composition. The notion of "vertical composition" consists in having a sub-component encapsulated in a composite component (irreflexivity applies in order to avoid any cycle). The latter hides the sub-component from clients and, more precisely, from the clients' service requests.

The implementation of vertical composition within *PauWare* relies on the theoretical research results exposed in [11-13]. In these three papers, a formal

[3] The second specificity of the UML 2 executability semantics is a special strategy for coping with conflicting transitions in statecharts. We do not address these issues in this paper. In short, nested transitions linked to inner states override upper transitions linked to outer states.

semantics for the *Aggregation* and *Composition* UML relationships is provided. A key feature of the UML *Composition* is coincident lifetime (a.k.a. "the fifth case of lifetime dependency" in [11]) between attached composites and compounds. This property of coincident lifetime makes the possibility realistic (and even relevant) that a component instance strongly refers to the states of another component instance. In comparison, the states of two different component instances do not have to be interrelated if these two components have unrelated lifecycles[4].

Going back to the *Railcar system* case study, one may thus consider that in terms of states, a single *Control center* component instance is a composition of all existing *Railcar* and *Terminal* component instances that participate in the system. In other words, following the logics of coincident lifetime assigned to *Composition* in UML, *Railcar* and *Terminal* component instances do not have to exist out of the life span of the *Control center* component instance. In terms of behaviors, a control center propagates or delegates environment data coming from sensors (*e.g.*, *alert100*, *alert80*) to railcars and terminals.

The proposed solution is based on the metamodel in Fig. 4. The *Whole-Part*, *Aggregation* and *Composition* types come from [11]. The right hand side of Fig. 4 is new and shows that the *Statechart monitor* type (embodying a global state machine) inherits from the *Statechart* type. In other words, a state machine is a kind of macroscopic state. However, the *Statechart monitor* type has interpretation capabilities as well: it possesses the *run_to_completion* Java method which is not owned by the *Statechart* type.

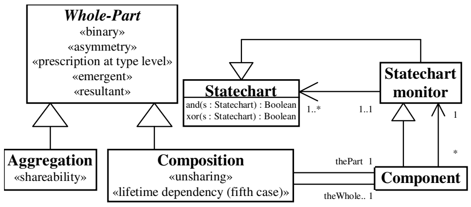

Fig. 4. UML metamodel for vertical composition

This leads to adding a specific service for the *Railcar* component type whose implementation is as follows:

```
public Statechart_monitor state_machine() {return _Railcar;}
```

Vertical composition is then instrumented as follows:

```
Statechart _Control = new Statechart("Control");

_Control_center = new
Statechart_monitor(_Control.and(railcar_remote.state_machine()),"Cont
rol center");
```

The code above illustrates the linking of a *Railcar* component instance state machine as a sub-state of the *Control center* component instance and as an orthogonal state of the *Control* sub-state: use of the *and* operator.

[4] In the worst case, two component instances may be connected together through their states but, in our opinion, with great care, since it is an error-prone situation.

Such a solution creates an automatic propagation/delegation mechanism. So, requests are forwarded from composites to compounds in a transparent way. In the code below, a multicast mode for sending the *alert80* request is used. All attached railcars, like the *railcar_remote* J2EE object in the code above, are concerned with the reception and the possible processing (depending upon their current state) of *alert80*:

```
public void alert80() throws Statechart_exception
{_Control_center.run_to_completion();}
```

In this code, no other processing except propagation/delegation occurs.

6 Conclusion: Perspectives and Benefits from an Enhanced Composition Model for Conversational EJBS

A side effect of having state machines inside components is the possibility for instrumenting dynamical re-configuration. For varied reasons, one may decide to force the state of a component. Externally, this leads to offering and to implementing a management service such as for instance *reset*:

```
public void reset() throws Statechart_exception
{_Terminal.to_state("Idle");}
```

Decisions may be taken by the components themselves. In this case, they become self-configuring and self-managing software entities, a concept of autonomic computing [14]. In this line of thought, a more advanced feature of autonomic computing is self-healing. *PauWare*'s components may support self-healing in the sense that the execution of any business request may generate faults. Fault self-management consists then in trying to “cancel” faults automatically. At this time, the implemented mechanism is rudimentary. When a fault occurs and if the autonomic mode has been activated, conversational EJBs try to recover their immediately previous “state”: this amounts to multiple consistent states, since statecharts are composed of nested and parallel modeled/implemented states. Within the cycle of moving a statechart from one step to another, internal operations in components may change business data (just before the arrival of the “incriminated” fault). Going back to the immediately previous state may therefore lead to inconsistencies. For example, a requirement may be that a port must be closed in a given state. Returning to this state without having the port closed is inconsistent.

To improve such a situation, state invariants that may be attached to states, have the responsibility to check if the current values of business data are compliant with the reached states. Within the process of fault recovery, this leads to proving that returning to the immediately previous global state of a component is “correct”. To sum up, self healing really succeeds if and only if all state invariants are true after rolling back to such an immediately previous state.

We have presented in this paper a concrete implementation of an enhanced composition model for conversational EJBs. A main motivation relating to such a research work, is the look for a better integration of the EJBs’ technology within MDA. One especially tries to fill the gap between two execution semantics. Model executability in UML is somehow idealistic but it benefits from being abstract, *i.e.*, independent of technological platforms. Instead, execution constraints of EJBs may be considered as numerous. Nevertheless, this is the source of a robust, but limited, composition model. A tradeoff is thus required, a goal of this paper.

The proposed implementation also favors the creation of a support for autonomic computing. Short-term perspectives are the adaptation of *PauWare* for J2ME components, *i.e.*, components that are deployed and run in mobile and wireless devices. This implementation, currently in its testing phase, aims at being in better convergence with the demands of autonomic computing.

References

1. Szyperski, C., Gruntz, D., Murer, S.: Component Software – Beyond Object-Oriented Programming – Second Edition, Addison-Wesley (2002)
2. Mellor, S., Scott, K., Uhl, A., Weise, D.: MDA Distilled – Principles of Model-Driven Architecture, Addison-Wesley (2004)
3. Bock, C.: UML 2 Composition Model, Journal of Object Technology, Vol. 3, 10 (2004) 47-73
4. Harel, D.: Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, Vol. 8 (1987) 231-274
5. Sun Microsystems: Enterprise JavaBeans™ Specification, Version 2.1 (2003)
6. Simons, A.: On the Compositional Properties of UML Statechart Diagrams, Proc. 3rd Conf. Rigorous Object-Oriented Methods (2000) 4.1-4.19
7. Prehofer, C.: Plug-and-play composition of features and feature interactions with statechart diagrams, Software and Systems Modeling, Vol. 3, 3 (2004) 221-234
8. Harel, D., Gery, E.: Executable Object Modeling with Statecharts, IEEE Computer, Vol. 30, 7 (1997) 31-42
9. von der Beck, M.: A structured operational semantics for UML-statecharts, Software and Systems Modeling, Vol. 1, 2 (2002) 130-141
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: An Open Component Model and Its Support in Java, Proc. 7th International Symposium on Component-Based Software Engineering, LNCS #3054, (2004) 7-22
11. Barbier, F., Henderson-Sellers, B., Le Parc-Lacayrelle, A., Bruel, J.-M.: Formalization of the Whole-Part Relationship in the Unified Modeling Language, IEEE Transactions on Software Engineering, Vol. 29, 5 (2003) 459-470
12. Tan, H. B. K., Hao, L., Yang, Y.: On Formalization of the Composition Relationship in the Unified Modeling Language, IEEE Transactions on Software Engineering, Vol. 29, 11 (2003) 1054-1055
13. Barbier, F., Henderson-Sellers, B.: Controversies about the Black and White Diamonds, IEEE Transactions on Software Engineering, Vol. 29, 11 (2003) 1056
14. Kephart, J., Chess, D.: The Vision of Autonomic Computing, IEEE Computer, Vol. 36, 1 (2003) 41-50