OBJECT-ORIENTED INHERITANCE OF STATECHARTS FOR CONTROL APPLICATIONS [1]

João M. Fernandes * Ricardo J. Machado **

* *Dept. Informática*
** *Dept. Sistemas de Informação*
Escola de Engenharia, Universidade do Minho
4700-320 Braga, Portugal

Abstract: This paper discusses how object-oriented inheritance can be re-interpreted if statecharts are used for modelling the dynamic behaviour of an object. The support of inheritance of statecharts allows the improvement of systems' development by easing the reutilization of parts of already developed successful systems, and by promoting the iterative and continuous models' refinement advocated by the operational approach. Statechart is the formalism used within UML to specify reactive state-based behaviours. This paper covers the use of statecharts within the modelling of embedded systems for industrial control applxications, where performance and memory usage are main concerns.

Keywords: Statecharts, Software engineering, Embedded systems, Control applications, Computer control system design.

1. INTRODUCTION

UML (Unified Modeling Language) is a general purpose modelling language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modelling and other non-software systems (Booch, *et al.*, 1999). UML is the OMG standard notation for defining and designing software systems, and is being progressively accepted in industry. UML is meant to be used universally for the modelling of systems, including automatic control applications with both hardware and software components (Douglass, 1997).

Within UML, statecharts are state-oriented models used to specify the dynamic behaviour of some classes, mainly those that show an interesting or complex behaviour. Statecharts are a visual formalism to describe states and transitions, enabling clustering, orthogonality, concurrency, and refinement and encouraging zoom capabilities for navigating between levels of abstraction (Harel, 1987). Statecharts add some useful extensions to flat Mealy-Moore state diagrams, such as nesting of states, conditional event responses, orthogonal regions and history connectors. Although these extensions are mathematically equivalent to Mealy-Moore state machines, statecharts can be made simpler and easier to read, especially for complex systems.

In object-oriented approaches, objects are instances of classes, which specify the attributes and the operations of its objects. The classes are related by hierarchical associations and, usually, inheritance means that a subclass can use the attributes and the operations of its superclasses. However, the use of statecharts (or other state-based formalism) to model the behaviour of classes introduces new problems, which are addressed by this paper that shows how inheritance among classes can be re-interpreted. The use of

[1] Partially supported by the Portuguese Science & Technology Foundation project PRAXIS/P/EEI/10155/1998, *Reconfigurable Embedded Systems: Development Methodologies for Real-Time Applications.*

statecharts is analyzed within the modelling of embedded systems for control applications, where performance and memory usage are main issues for the final system.

2. CODE GENERATION

The mapping of a class' statechart into code is divided in two phases. Firstly, the statechart behaviour is mapped to classes properties (attributes and operations) and then the class is transformed into code. This section presents some guidelines that support the first mapping phase.

The mapping of statecharts into code is not trivial, due to the numerous modelling mechanisms that exist in the statechart formalism. The most common software implementation of a statechart is with an enumerated variable that is used as a selector of a switch command (Booch, *et al.*, 1999; Douglass, 1998). Each case clause implements a given state of the diagram.

This approach is conceptually simple, but becomes problematic when applied to relatively complex state machines, especially models that are hierarchically described. In this case, the "flat" state machine must be considered, which usually means an exponential growth on the number of states and transitions (Selic, *et al.*, 1994). Thus, this implementation technique is not efficient, since complex systems must be considered.

Several authors presented proposals that suggest the way code must be generated from state machines that model the internal behaviour of objects. Among those proposals, the 'State' (Gamma, *et al.*, 1995) and the 'State Table' (Douglass, 1998) design patterns and the solution presented by Shlaer and Mellor (1992) have to be taken into account.

Those solutions were mainly idealized for software systems where principles such as reuse, extensibility, and simplicity are usually more important than performance and memory usage. However, for real-time embedded systems, these last two issues are quite important, since temporal deadlines and space restrictions (in terms of memory, for example) must be considered. With this perspective, the selected solution must differ from the previous ones and follows the main ideas presented by Metz, *et al.* (1999), with minor changes.

Using fig. 1(a), admit that the sound attribute and the adjustSound operation are already defined and that it is wanted to reflect the statechart behaviour at the corresponding class. Since adjustSound is independent of any object's state, it can be invoked at any time during the object's life cycle, so it is not represented in the statechart.

During the first mapping phase, the varState attribute is added and for each state one method is also added to the class. For each event, a method is again introduced into the class, which will be called whenever that event is detected. Finally, for each action and activity on the statechart, a class method is included. Fig. 1(b) depicts the class after introducing the attributes and operations that originate from the statechart.

To better organize the attributes and operations, UML stereotypes can be used to classify each group (Booch, *et al.*, 1999). Thus, the usage of the following stereotypes can be helpful to indicate the origin of the operations: ≪state≫, ≪event≫, ≪action≫, ≪activity≫, and ≪global≫ (for operations independent of the statechart).

Statecharts may refer variables, which can be used, for example, to reduce the number of states. Those variables must be added to the class as attributes. For example, in fig. 6 the readyCD attribute used in the statechart must be added to the corresponding class.

Whenever an object receives a message or an event, the corresponding method must be called. The method verifies if that message can be accepted in the actual state (value of varState). If the message is accepted, the transition condition is tested and the actions are executed. Then, the target state method must be invoked, where the value of the varState attribute is updated to refer to the new state.

3. INHERITANCE OF STATECHARTS

In object-oriented modelling, the characteristics of a class are influenced by all its superclasses. Thus, when an object behaviour is defined by a statechart, inheritance issues among the classes must be considered. A solution to this problem is to completely ignore the superclasses' statecharts and draw a completely new statechart (possibly with a totally different structure) for the class. This solution is not considered, since it is contrary to the object-oriented modelling principles.

The statechart of a class must be inherited by its subclasses. For this purpose, some rules, similar to those that are used for code inheritance, must be fulfilled. In this section, some of those rules and the respective problems are presented.

Due to the nature of the graphical modifications that can be done to a class' statechart, it is not possible to indicate those modifications in a simple and incremental way. Embley, *et al.* (1992) present a notation that allows a state model to be incrementally specified taking into account the superclass' statechart. This notation reduces the likelihood of making errors and inconsistencies, since models are built in a differential way which highlights the differences between the generic and the specialized behaviours. Nevertheless, the approach can not be applied generically. It is only

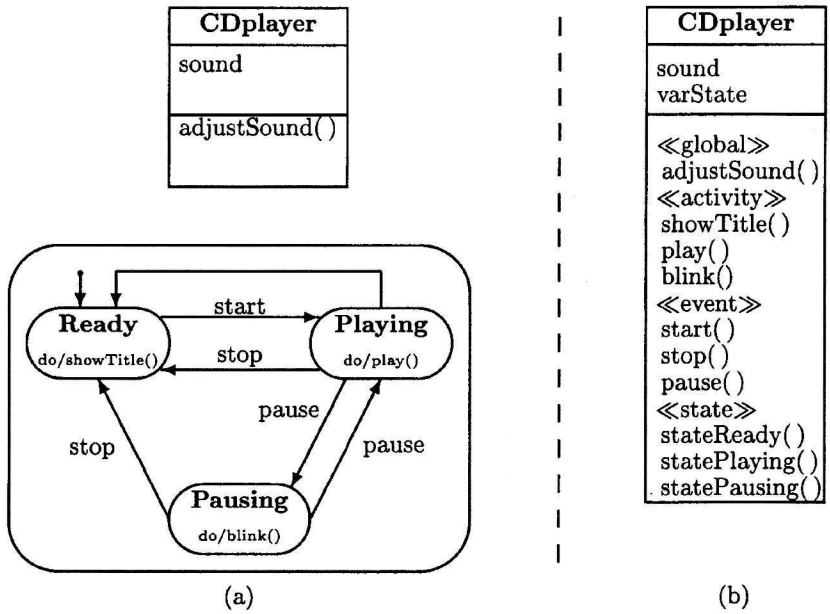

Fig. 1. The way (a) a class and its statechart (b) are merged.

usable when the modifications are limited and localized, since the topological location of the modelling elements must be preserved.

Our proposal is to consider the statechart of a class to be graphically a complete diagram. There are some proposals to highlight the modifications made to a class statechart in relation to the superclass' statechart. Weber and Metz (1998) suggest dashed symbols for inherited elements and normal symbols for the new elements, while Selic, *et al.*, (1994) use gray symbols for inherited elements and black symbols for the new elements. Whatever notation selected, the user must always draw a completely new statechart, which means that if some modifications are made to a class' statechart diagram, they must also be made (by hand and not automatically) to all its subclasses.

Another alternative is to transform the statechart into code (through pre- and post-conditions and operations), which allows the statecharts to be inherited through the usual mechanism available in all object-oriented languages (Oblog, 1997). The main drawback of this solution is that the graphical view of statecharts is lost, but, this way the designer can incrementally specify statecharts.

Inheritance of statecharts at the graphical level are addressed in this paper. Table 1 presents 8 rules that can be applied to a class' statechart to use it to specify the behaviour of a subclass. The proposals of different researchers are analyzed, which allowa the selection of the most relevant ones for control applications. Inheritance can be used for distinct purposes (subtyping, strict inheritance, refinement) and this work considers strict inheritance, which implies that properties can be replaced or added, but not deleted.

Before presenting the rules, let us answer the following question: what does it mean an object of class A being also an object of (a more generic) class B? In the majority of the approaches to inheritance, the "is_a" relation between classes A and B implies as a minimun requirement that A and B have the same interface. This means that a B's instance can be placed where an A's instance can also be placed, as long as the B's interface is consistent with the A's interface.

This requirement does not necessarily imply that there is behavioural consistency between both classes. It only guarantees that A's instances can be replaced by B's instances without incompatibilities, but it can not assure that objects B will behave the same way as objects A do. It is possible that objects B will react differently in relation to the behaviour of objects A. In practical terms, it is technically difficult to guarantee the behavioural compatibility between two classes (Harel and Gery, 1997).

However, usually there is no need to impose that the relation between a class and one of its subclasses means also that they do the same and in the same way. Normally, we want B to answer the same stimulus that A reply, but not necessarily in the same way. This signifies that inheritance is introduced mainly for allowing reuse.

3.1 *Unmodified statechart*

Whenever attributes and operations that have no dependency on the object's state are added to the subclass, the superclass' statechart can be used without any change to describe the subclass behaviour. This implies that the added properties

Rule	A	B	C	D	E	F	G
h1. Unmodified statechart	yes	yes	yes	yes	yes	yes	yes
h2. Redefinition of state activities	yes	yes	no	yes	yes	yes	n.d.
h3. Adition of transitions and states	yes	yes	yes	yes	yes	yes	restrited
h4. Change transitions' target state	yes	yes	no	no	yes	yes	no
h5. Removal of transitions	no	yes	no	no	partial	no	no
h6. Refinement of the transitions labels	yes	yes	yes	yes	yes	yes	n.d.
h7. Removal of states	no	yes	no	no	no	no	no
h8. Change transitions' origin state	no	yes	no	no	no	no	no

Table 1. Inheritance rules and how different methods support them. A (Douglass, 1998), B (Selic, *et al.*, 1994), C (Shlaer and Mellor, 1992), D (Embedded systems), E (Weber and Metz, 1998), F (Harel and Gery, 1997), G (Rumbaugh, *et al.*, 1991).

can be used anytime during the object's life-cycle. This can also be thought as the addition of orthogonal behaviours to the statechart.

For example, if in a CDplayer subclass, the attribute price and the operation activateFilter are added and if they have no dependency on the object's states, the superclass' statechart of fig. 1(a) can be used without modifications to describe the behaviour of the subclass.

3.2 *Redefinition of state activities*

In a subclass, it must be possible to redefine (i.e. specialize or add) the activities and the actions associated with a state. Douglass (1998) allow actions and activities from a state to be removed, without any restriction. This possibility must only be considered if the operation associated to the removed action or activity is still referred in the statechart. If this happens, the operation is still valid in the subclass. If all the operation references are removed, it is possible by inheritance to use an operation at the subclass whose access was not considered, which may cause inconsistencies.

This rule is illustrated in fig. 2, where the activity associated with the state Ready is modified in the subclass. The showInfo activity is understood as a specialization of showTitle, and must be defined in the subclass. Within this rule, it would be also possible to make modifications to (entry and exit) actions associated with states.

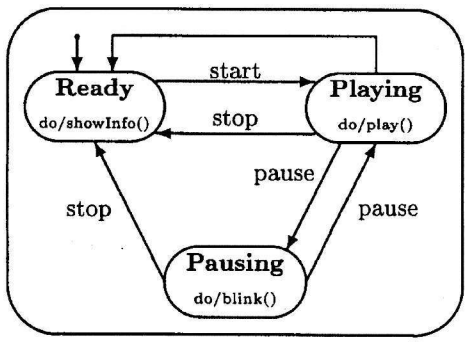

Fig. 2. Statechart with a modified activity.

3.3 *Adition of transitions and states*

A subclass may add states and transitions to the superclass' statechart. This possibility is related to the code inheritance mechanism used to add attributes and operations to the subclass. Adding a new state forces new transitions linked to inherited states to be also added and (new or inherited) events to be associated with those transitions.

Thus, adding new transitions to inherited states is valid, since it is a particular case of this rule. However, to build a deterministic statechart, it must be assured that the guards associated with all the transitions that originate from a given state are mutually exclusive. This aim may imply that the guards associated with the inherited transitions must be redefined.

The term "adition" of this rule must be interpreted in a broad sense. The proposals by Douglass (1998) and by Harel and Gery (1997) include the following changes to a state, whose practical consequence is a statechart with more states: (1) Decompose a state into concurrent sub-states (orthogonal components); (2) Add one sub-state to a concurrent state; (3) Add orthogonal components to a sequential state.

The OMT proposal (Rumbaugh, *et al.*, 1991) is the most restrictive one, since it does not allow the direct introduction of new states and transitions, in the superclass' statechart, since the latter must be a projection of the subclass' statechart, that is, this must be a refinement of the former. Thus, any state of the superclass' statechart may by specialized or divided into concurrent parts.

It is also allowed to introduce new transitions triggered by new events. Fig. 3 illustrates an example of this situation with the introduction of a transition labelled with a new event (go). This rule also allows the introduction of a new state (Changing) to change the track actually being reproduced. This example shows that the introduction of a new state implies the introduction of transitions (in the example, enabled by new events) that connect the new state to some of the inherited states. During the code generation process, it will

be necessary to include new methods for the go event, the Changing state and the change action.

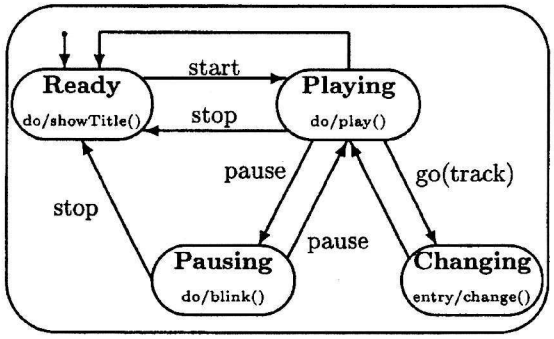

Fig. 3. Statechart with a new state, a new event, a new activity and new transitions.

3.4 *Change transitions' target state*

Changing the target state of a transition may be allowed, as long as it is possible to reach the previous target state by a different transition. If this is considered, it is guaranteed that the activities and actions of the target state are still needed in the subclass' statechart.

The application of this rule is usually necessary when a new state is introduced between two inherited states. In this case, the new state becomes the target of the transition that previously (i.e., in the superclass) linked the inherited states and a new transition is added between the new transition and the target state of the modified transition.

In fig. 4, the Reading state was introduced, allowing the CD player to read a program that selects which tracks will be reproduced. This example shows how a new state (Reading) can be introduced between two inherited states (Ready and Playing) and the change of the target state of the transition labelled with the start event.

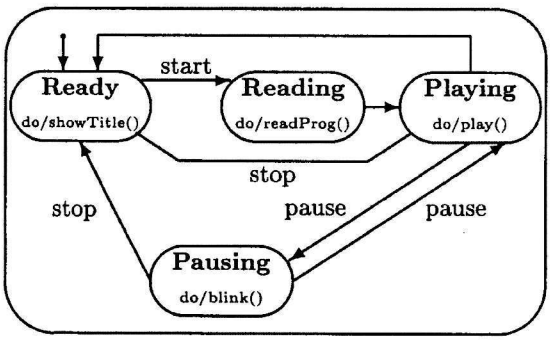

Fig. 4. Statechart with a new state between two inherited states.

3.5 *Removal of transitions*

The removal of a transition does not produce any inconsistency in terms of the operations available at the subclass, as long as at least one subclass' transition has as its event the one that is associated with the removed transition. If all the transitions associated with a given event are eliminated, the subclass can still refer that event (through inheritance) which can cause an inconsistency. In this sense, the removal of automatic (unconditional) does not pose any problem and can be performed. Notice that the removal of a transition may imply the redefinition of the guards associated with the transitions that originate from the same state as that of the removed transition.

A transition can be logically removed by redefining its guard condition to have the false value (through rule h6). With this facility, the transition is still represented in the diagram, but is never fired since the condition is always false.

Fig. 5 exemplifies this rule. In the subclass' statechart the transition that linked states Pausing and Ready was removed. In this case, it is impossible to stop the CD player, whenever it is paused. This transition removal is allowed, since there still exists a reference to the stop event in the subclass' statechart. In the subclass, it is necessary to rewrite the method of the stop event, since it must now be ignored when the actual state is Pausing.

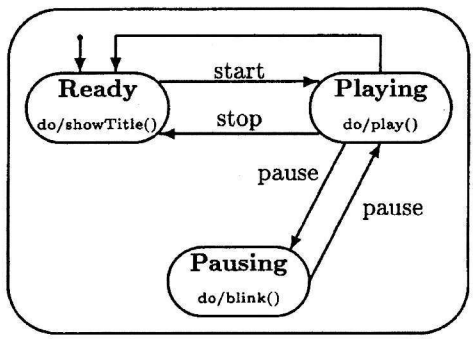

Fig. 5. Statechart with a removed transition between 2 inherited states.

3.6 *Refinement of the transitions labels*

Transitions are labelled with inscriptions that follow the format: *Event[Guard]/Action*. Labels can be specialized, meaning that the subclass may modify (i.e. add, eliminate or change) the event, the guard condition or the actions. As stated in rule h5, if the condition of a transition is changed to false, that transition is being logically removed.

The use of this rule is shown in fig. 6, where the label of the transition that links states Ready and Playing is changed. In this example, the condition [readyCD] is added, which refines the way the transition is enabled to fire. This implies that the readyCD attribute must be added to the subclass. Within this rule, other examples where new actions are added to a transition could also be considered.

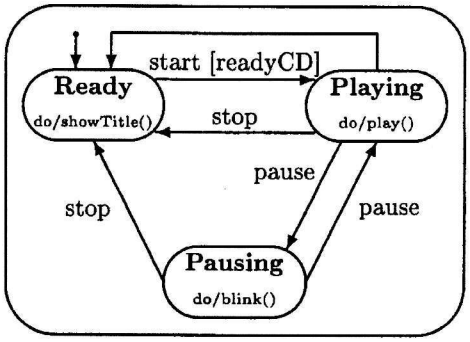

Fig. 6. Statechart with a redefined transition.

3.7 *Removal of states*

The removal of a state should be disallowed since the subclass looses access to a state that exists in its superclass' statechart. If a state is actually removed, an inconsistent situation may be created since the method associated with that state is defined at the superclass and thus available by inheritance.

A way of avoiding this problem is to logically eliminate all the transitions that end at that state, by redefining their guards to be false (applying rule h6). Consequently, the state is removed since it is not connected to other states. Although it is still graphically represented, in pratical terms it was removed since it is unreachable.

3.8 *Change transitions' origin state*

Usually, it is forbidden to change the origin state of a transition. Although this rule is not generally allowed, there seems to be no firm reason for that. Anyway, it is also not authorized in this work, since its application introduces a conceptual problem contrary to the object-orientation principles.

4. CONCLUSIONS

This paper showed how object-oriented inheritance can be re-interpreted when statecharts are used to model the dynamic behaviour of a class. The presentation is oriented to the use of statecharts within the modelling of embedded systems for industrial control applications, where issues such as performance and memory usage are more important than reuse, extensibility, and simplicity.

The mapping of a class' statechart into code is divided in two phases. Firstly, the statechart behaviour is mapped to classes properties and then the class is transformed into code. In this paper, some rules and guidelines that support the first mapping phase are discussed. Eight inheritance rules for statecharts were discussed in what concerns its acceptance or rejection for the development of embedded control systems.

In this work, strict inheritance among classes was considered. Since inheritance can be used for different purposes (subtyping, strict inheritance, refinement), it is important to realize that the rules presented here may not be directly used and should be adapted if a different purpose is selected. The approach followed is specifically tuned to help the development of control embedded systems, so it corresponds only to a particular domain solution.

REFERENCES

Booch, G., J. Rumbaugh and I. Jacobson.(1999). *The Unified Modeling Language User Guide.* Object Technology. Addison-Wesley.

Douglass, B.P. (1997). Designing Real-Time Systems With The Unified Modeling Language. *Electronic Design*, pp. 132–42. September.

Douglass, B.P. (1998). *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Object Technology. Addison-Wesley.

Embley, D.W., B.D. Kurtz and S.N. Woodfield. (1992). *Object-Oriented Systems Analysis: A Model-Driven Approach.* Yourdon Press.

Gamma, E., R. Helm, R. Johnson and J.M. Vlissides. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing. Addison-Wesley.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–74.

Harel, D. and E. Gery. (1997). Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42.

Metz, P., J. O'Brien and W. Weber. (1999). Code Generation Concepts for Statechart Diagrams of the UML v1.1. In *Object Technology Group Conference*, Vienna.

Oblog Software. (1997). Oblog Language Guide. Technical report, Oblog Software S.A.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. (1991). *Object-Oriented Modeling and Design.* Prentice-Hall.

Selic, B., G. Gullekson and P.T. Ward. (1994). *Real-Time Object-Oriented Modeling.* John Wiley & Sons.

Shlaer, S. and S. J. Mellor. (1992). *Object-Lifecycles — Modeling the World in States.* Prentice-Hall.

Weber, W. and P. Metz. (1998). Reuse of Models and Diagrams of the UML and Implementation Concepts Regarding Dynamic Modeling. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language: Technical Aspects and Applications*, pp. 190–203, Heidelberg, Physica-Verlag.