Extending Statecharts for Representing Parts and Wholes

Luca Pazzi
Dipartimento di Scienze dell’Ingegneria
Università di Modena
Via Campi 213/B, I-41100 Modena - Italy
Ph.: +39-59-378516 - Fax: +39-59-370040
E-mail: pazzi@unimo.it

Abstract

As state-based formalisms and object-oriented development methods meet, Statecharts represent a natural choice for object behavioural modelling. This is essentially due to built-in features that enforce modularity and control complexity. The paper suggests to improve the effectiveness of the Statechart approach in achieving both modularity and reuse of behavioural abstractions by analysing the general problem of modelling parts and wholes. An extended Statechart construct is proposed, which improves the capability of separating global from local contexts in the early phases of the object development process, getting better global software quality factors.

1. Introduction

The problem in the specification and design of complex systems is rooted in the difficulty of describing their behaviour in a way that is at the same time clear, realistic, formal and rigorous [10]. State-based formalisms (for example Finite States Automata [12]) provide for specifying behaviour of complex systems in terms of the complementary concepts of states and state transitions: the approach presents advantages as well as problems. The advantages of using state-based formalisms is that the concept of state provides an intuitive notion of the system being modelled (states are interpreted as “snapshots” of the situation at a given point in time), which can be subsequently refined and formally anchored to the attributes of the objects [2]. The problem of using state-based formalisms is related to the intrinsic complexity of the real world. As the dimension of the problem increases, state transition diagrams become exponentially unmanageable. The state explosion problem requires controlling the overall complexity by partitioning the whole state machine by different state machines thus reducing the number of states and state transitions.

Statecharts, by David Harel [7][8], provide valuable features to permit concise descriptions of complex systems, since state decomposition supports — in a natural way — stepwise, top-down and bottom-up development. States are, in the Statechart setting, the basic modular construct for software engineering tasks. State decomposition closely follows object specialisation and generalisation by nesting states, and object aggregation by orthogonal states.

The object-oriented paradigm supplies the natural context for providing Harel’s state machines with a defined boundary, by using objects not only to represent structure, but also to localise behaviour [6]. As objects partition the world, flat state transition diagrams are decomposed into meaningful and manageable units, defeating the overall complexity of real world.

The crucial aspect is therefore the encapsulation of behavioural aspects within the object boundary. A proper encapsulation starting from the early phases of the development process impacts on the whole development process [3], yielding abstractions which are reusable, understandable and extendible [13]. The problem of finding the effective object-boundary for objects and their behaviour requires not only looking at the bare individuals, but also at the way they compose in making more complex entities: this way, we propose to consider the issue as part of the more general problem of the explicit modelling of wholes [1]. A whole represents a structural and behavioural abstraction inferred from associative knowledge in the domain. We propose to enforce object encapsulation in Statecharts by the explicit recognition of wholes from associative behaviour knowledge, essentially synchronisation. Composition among different Statecharts is usually achieved through synchronisation of orthogonal state machines by the broadcast mechanism. Synchronisation can be modelled either implicitly, by allowing mutual references between orthogonal state machines, or explicitly, by extracting synchronisation aspects from the state machines being synchronised and encapsulating them into an additional
state machine, representing the whole, which nicely caters for reuse and extendibility.

The paper proposes firstly to recognise the implicit/explicit distinction as appropriate in the context of compositional behavioural modelling, and secondly to adopt a revised form of state modelling formalism, with the aim of enforcing the explicit approach. The work is structured as follows: Section 2 presents and discusses the basic mechanism by which Statecharts enforce modularity, Section 3 discusses the problems with the current, implicit way of achieving composition by orthogonal state machines and presents an explicit approach for Statecharts: the proposal is further refined in Section 4 by illustrating a modelling example.

2. Statecharts = state decomposition + broadcast mechanism

2.1 Modularity in behavioural specification

State decomposition and event broadcasting contributed in making Statecharts a successful modelling tool in object-oriented development methods [5]. Beyond the reduction of the overall complexity of the problem, it is interesting to analyse the success of the Statechart approach as related to the enforcement of software quality principles, such as encapsulation. As an example, Harel observed [10] that the limited and localised effect of the chain reaction of the events triggered by a state transition (figure a) contributed to ease the specification of complex internal behaviour. The same author observed also that there is no direct relationship between the broadcast mechanism of Statecharts and traditional inter-object communication, thus leaving room for additional research on the way parallel parts of AND-decomposed state machines interact and synchronise, in order to explore the potentiality of state and object composition and reuse. The idea pursued by the paper is that the specification of complex systems can be substantially improved by directly enforcing software engineering principles through a disciplined use of both state decomposition and event broadcasting (Section 3.3).

2.2 State decomposition

State decomposition is the most important extension to traditional state transition diagrams. It is the ability to decompose states, both by nesting states through exclusive-or decomposition (an object in an XOR decomposed state Z can be found either in state A or B, but not in both) and by decomposing states through parallel AND decomposition (an object found in a AND decomposed state is at the same time in each of the states resulting from the decomposition). State decomposition arranges states in a parent/child relationships. Starting from a root state, the states may be arranged in a tree of XOR- and AND- related nodes. Each time the object is found in one or more leaf nodes of the tree. The global state of the object is the set of states which are collected starting from the root to the leaf.

XOR-decomposition. A parent state (super-state) contains different child states (sub-states). In such case, entering the super-state means entering - at the same time - exactly one of the sub-states. This kind of decomposition means therefore “refining a snapshot”, by identifying sub-aspects which refines the state description.

AND-decomposition. In this case entering the parent state means -at the same time - entering all of the child states. Each of the sub-states may be further decomposed in either the AND- or XOR- fashion. By an AND-decomposed state we specify that a system is made of parallel, orthogonal aspects, i.e. aspects that may be taken simultaneously. AND-decomposed states provided the idea for orthogonal state machines, i.e. machines in which the total number of states is made up from the Cartesian product of states from different machines: using orthogonal state machines reduces the global number of states and, above all, allows to compose and reuse behavioural abstractions.

2.3 Broadcast mechanism

As observed in [7] one of the major difficulties in reducing the state complexity of the domain by partitioning it into separate state machines is the “need to cater for synchronisation and communication among the resulting state machines”. Hierarchical and parallel state decomposition provide a straightforward rule for propagating events along decomposed states. An informal syntax and semantics for the subsequent discussion is briefly outlined in the following; for a deeper discussion on the semantics of Statecharts see [9].

A state transition \( t: e [C] / a \) is identified by a label \( t \) and composed by three optional parts: a triggering event \( e \), a condition \( C \) and an action part \( a \). The action part is a semicolon-delimited list of action clauses \( a_1; a_2; \ldots; a_n \) each specifying either an event to be propagated (see for example \( f \) in the next example) or a statement assessing a change in the instance variables of the object (see figure g for an example).
Figure A. Chain reaction for the broadcast mechanism. An event $e$ triggers state transition $t$ which, in the subsequent step, broadcasts event $f$.

The semantics of event propagation is the following (figure a): the occurrence of an event $e$ in a state decomposed by the parallel sub-states $R$, $S$, and $T$ causes the transition which has $e$ as triggering event to be taken depending on the associated condition $C$. The action part of the transition is immediately taken and is regarded as a new event $f$ to be broadcast. Observe that the propagation of events is directed towards the nested and orthogonal components of a single state which therefore acts as a boundary for the “chain reaction” triggered by the occurring of an external event.

![Figure A. Chain reaction for the broadcast mechanism.](image)

Figure B. Parallel, not synchronised aspects described by orthogonal machines.

3. Modelling Concurrent and Synchronised Behaviour

AND-decomposed states provide a powerful tool for expressing parallel aspects of the object behaviour. Such aspects evolve concurrently and occasionally interact by synchronising on some event sequences. The two cases of independent and synchronised concurrent evolution may be illustrated by adapting an example from [6]. In the first case (figure b) a Bottle carried by a conveyor belt in a filling plant is clamped in a position and filled with a liquid (the two aspects are independent and parallel). It is therefore possible to express the compound behaviour “being on the conveyor belt” in separate Statecharts, each expressing the specialised behaviours “bottle clamping” and “bottle filling”. Sticking them in a final AND-composed state yields the global behaviour. More realistically however, we need to express some synchronisation constraints among the aspects making a complex behaviour. For example, a minimum requirement is that the Bottle is required to be clamped before being filled (figure c). Synchronisation between the different parts of an AND-decomposed state may be achieved in different ways. For example it is possible either to guard the fill transition in the Filling part of the Statechart or by requiring that the same transition happens in both the Filling and Clamping part.

3.1 Interaction and synchronisation in Statecharts

The different parts of AND-decomposed state machines interact and synchronise by the combined effects of the broadcast and triggering mechanisms. Synchronisation among parallel parts of Statecharts is achieved by sharing and triggering events and by testing mutual conditions.

- **event sharing**: the same event name labels more than one transition in different parts of the decomposed state. The happening of the shared event, the different state transition are taken simultaneously in the different parts, making them to evolve in a parallel way. Event sharing is the core of the synchronisation approach in Hoare’s Communicating Sequential Processes (CSP) [11].

![Figure B. Parallel, not synchronised aspects described by orthogonal machines.](image)

![Figure C. Synchronising orthogonal parts of a Statecharts by (a) event sharing and (b) mutual conditions testing.](image)
• **mutual conditions testing**: before an action is taken, the condition part of the transition is evaluated. By referring to states or instance variables in a parallel part, the object behaviour depends on another object being in a given state.

• **event triggering**: as result of a transition an event is broadcast to the parallel parts of the Statecharts. The occurrence of an event \( e \) causes the transition labelled \( e / f \) to be taken. The *action part* of the transition is immediately “executed” and is regarded as the *occurrence* of a new event \( f \) (figure a).

The above approach in composing Statechart has been recognised and adopted by different object-oriented development methods for the purposes of analysis and requirement modelling. In Rumbaugh’s Object Modeling Tool (OMT) [16] it is observed that the state diagrams for the various classes combine into a single dynamic model via *shared events and guarded transition*.

### 3.2 Implicit versus explicit modelling of wholes

Composing Statecharts by the synchronising mechanisms depicted in figure c may be regarded as an *implicit way* of achieving the composition of a whole from its parts. Conceptual modelling of aggregate entities can be realised by two philosophies referred to, respectively, as implicit and explicit modelling of wholes [1]:

- By the *implicit* approach an aggregate is modelled through a web of references by which the components refer one to another. This way the *associative knowledge* between them is modelled directly, *hiding the structure and the behaviour of the whole* which is therefore *not* identified as a relevant entity in the modelling.

- By the *explicit* approach, an explicit *whole entity* is emphasised. This way, the *associative knowledge* between the components is not modelled directly, rather it reveals an aggregate entity as relevant in the modelling.

Although the distinction among an implicit and explicit way of composing whole entities from part entities was formerly devised in the object-oriented context, it may be applied to any module-based formalism: in the Statechart case the associative knowledge involved is essentially *synchronisation* [14]. Traditional, *implicit* modelling is represented by the way *synchronisation* is achieved by mutually referencing aspects. In [16] it is claimed that such mechanism allows interaction between the state diagrams, while preserving modularity. Consider instead that the above mentioned aspects break encapsulation, making the resulting abstractions barely reusable, understandable and extendible, thus jeopardising software quality factors [13]:

*reusability*: synchronising the two aspects through event sharing as in figure c-a, requires to share the same set of events (an alphabet of events in the CSP [11] terminology). This makes difficult reusing the clamping behaviour in other parts of the bottling plant. For example representing a compound behaviour related to the labelling phase, requires to add a third transition to the “stopped” state. Conversely, by adopting the solution depicted in figure c-b, any change in the left part of the diagram would require corresponding changes in the left part.

*understandability*: the problem in understanding an implicit complex behavioural abstraction is that we have to look at many different parts in order to understand the whole behaviour.

*extendibility*: extending an implicit description is difficult since we have to modify different parts in order to add a single whole feature. Section 4.4 discusses extension examples.

Observe therefore that composing Statechart by the implicit approach, calls for *tightly coupled abstractions* by breaking encapsulation and hiding more complex wholes. If, on the other hand, the *structuring role of synchronisation* as associative knowledge is properly recognised, an explicit whole entity is inserted in the modelling with the explicit role of encapsulating coordination and synchronisation aspects. In short, recognising the explicit associating role of synchronisation means that whenever different entities are synchronised, the result is an additional entity, having the synchronised entities as parts.

### 3.3 An explicit modelling approach for Statecharts

As observed, Statecharts are a successful modelling tool since they help in enforcing object-oriented design principles. However, it can be observed that a substantial evolution might result from the *disciplined use* of the standard mechanisms already available in the Statechart framework. The idea is to force the modeller towards the explicit identification of *wholes*, thus achieving further improvement of software engineering quality factors. As structured programming is realised by a disciplined use of *gotos* (embodied in higher level constructs such as
iterative instructions), in the same way the modeller is forced towards a correct modelling of the domain by higher level state modelling constructs, which embody mutual intermodular references. Such whole-modelling constructs have a twofold impact:

- enforce encapsulation
- reveal wholes as explicit entities

Observe that the two aspects are closely related. A whole results from the features scattered in the different parts. As shown in Section 4, wholes may not be readily apparent during the analysis phase of the domain. Their explicit modelling, however, on one hand allows to identify reusable abstractions for the parts, on the other hand the whole itself is a meaningful, reusable, understandable and extendible abstraction.

Extending Statecharts for the explicit modelling of wholes means providing a specific section for the whole and different sections for the parts (figure d). The “whole” section and the “parts” sections evolve concurrently as in orthogonal state machines. However to enforce the software engineering principles of encapsulation, we refine event broadcasting by the following rules:

- Events cannot trespass the boundary among the different part sections. To emphasise this aspect, the line which separates the different part sections is drawn as solid.
- Events can be broadcast both from the whole section to the parts section and vice versa. To emphasise this aspect, the line separating the parts from the whole is drawn as dashed.

By prohibiting any communication and mutual knowledge among the different parts we enforce self-containment and encapsulation. The whole therefore knows its parts, which, on the other hand, ignore each another. Observe finally that extended Statecharts with empty “parts” sections have the nice property of reducing to regular Statecharts.

3.4 Naming mechanism

In the following a simplified naming mechanism is provided by adopting the following rules:

1. Each “part section” is identified by a part or role name.
2. Each Statechart feature (events and states) in each part is referred to by dot prefixing its name with the name of the parts in which it is contained.

For example (see figure i) the drop event in the nozzle section of the extended Statechart is referred to as nozzle.drop. The same convention applies to states.

4. An example

In this section we present an example showing two interacting entities: a nozzle drops tennis balls, which fill boxes carried by a conveyor belt (figure e). The point is that by providing the appropriate construct (i.e. by not allowing entities to synchronise directly and collecting the synchronisation aspects in a specific whole section) the analysis is forced towards a correct modelling of an additional whole entity, not readily observable at a first glance.

![Figure E](image-url)
Figure F. Statechart for the entity Nozzle.

The Statechart for the entity Box (figure g) presents three states of interest (Empty, Filling and Full) connected by the guarded event fill denoting the receiving of a tennis ball. A ball can be received as long as the Box is either in state Empty, or in state Filling. Each occurrence of a fill event within the Box increments the instance variable n which denotes the number of tennis balls received by the box. The constant MAX denotes the maximum number of balls contained in a single Box.

Figure G. Statechart for the entity Box.

4.1 Implicit Synchronisation

By the Statechart approach, the interaction between the two entities is represented in figure h by synchronising orthogonal parts of a Statechart. As observed, an implicit approach may be carried out by different means, such as event sharing, mutual state testing and event triggering. In any case describing the interaction, requires to modify the original Statecharts by sticking mutual references (emphasised by dashed boxes in figure h).

For the interaction to happen, some preconditions must be met, and some effects are observed after it completion. The different parts of the interaction may be stated in the following terms.

- **Global pre- and post-conditions:** As observed, the start and the stop event in the Nozzle precede and follow any sequence of drop events. Observe that as long as the nozzle is taken in isolation the constraint is correctly described within the Nozzle Statechart (figure f). The extended global behaviour requires now to express that any sequence of drop and fill events is initiated by a start event and terminated by a stop event in the Nozzle. The (implicit) modelling of the extended constraint is achieved by adding an action part in the initial and in the final state transition of the Box Statechart. Observe that this way a local context (the Box Statechart) is utilised to express a constraint that is inherently global, thus breaking encapsulation.

- **Pre-condition:** before any drop event in the nozzle could take place, the box is required not to be in the Full state. This requires to test the state the counterpart is currently in, by sticking in the “precondition part” of the drop state transition an appropriate test (not in(box.Full)). It can be observed that this break the self-containment of the Nozzle Statechart.

- **Interaction:** Describing that any drop event in Nozzle corresponds to a fill event in Box is achieved by triggering a fill event in the “action part” of the drop state transition. By the broadcast mechanism, the fill event is propagated to the box Statechart. Again, observe that this way the Nozzle Statechart includes features belonging to the other Statecharts.

- **Post-condition:** the effect of the interaction (state variable n is incremented by each occurrence of the drop-fill interaction) is contained within the Box Statechart boundary.

The above specifications are summarised by the

Figure H. Implicit modelling of the composition of the nozzle with the box by orthogonally composed state machines. Features emphasised by dashed boxes break self-containment and encapsulation.
following synchronisation constraints:

C1: A start event in the nozzle precedes the entire interaction.

C2: A stop event in the nozzle follows the entire interaction.

C3: Any nozzle drop event is followed by a box fill event.

C4: The precondition for the drop event is that the box is not in state Full.

The discussion about software quality factors of Section 3.2 can be easily applied to the case of figure h. We briefly recall that reusability is jeopardised since the nozzle is no more a “stand alone” nozzle, rather it is a “nozzle which drops balls into a box”. As such it is not reusable outside of that context. The same can be observed for the box which is tightly bound to the nozzle. Observe also that embedding the global context (Nozzle+Box interaction) in the local context of Box and Nozzle makes them hardly understandable and extendible. The idea is therefore to make the global context explicit.

4.2 Explicit Synchronisation

The hypothesis guiding the research (Section 3.3) states that the explicit composition of state machines by “whole and parts” sections is equivalent to the implicit composition of state machines by orthogonal sections. Equivalence means that the same behaviour is described by the two cases. In this Section we empirically demonstrate the hypothesis by showing — in first place — that the synchronisation aspects singled out as constraint in Section 4.1, can be expressed entirely in the “whole” section, without affecting any of the original behavioural descriptions associated to the composing entities. In second place we analyse the result in terms of reusability, understandability and extendibility.

4.3 Modelling synchronisation in the “whole”

Constraints C1 and C2 are expressed by the initial and final state transitions labelled x and z in figure i. Observe that state transition z expresses the synchronisation constraint C4 by the precondition testing the state Box is currently in. Constraint C3 is finally modelled by specifying the sequence of drop and fill events in the action part of state transition z.

4.4 Extending behavioural descriptions

Finally observe that making explicit a “whole” section provides a natural context for extendibility. Suppose to extend the behaviour in order to fill the first half of the box with balls from a nozzle and the second half with balls from another nozzle. It can be observed that the specification is easily accomplished by adding a second nozzle section as in figure j. Consider how adding such details to the implicit modelling of figure h would have cluttered the diagram.

5. Conclusions

Although the object model is suited to uniformly represent data and procedural abstractions, most of the current approaches to object-oriented software development abstract independently structural and behavioural information during the analysis and design phase. A typical example of this approach is OMT [16] where an object modelling phase precedes a dynamic modelling phase. Dynamic model structure is thus related and constrained by object model structure. The only interaction allowed between the two kinds of modelling

Figure I. Explicit modelling of the composition of the nozzle with the box. Observe that a “black box” reuse of behavioural abstractions is suggested by reporting only the outline and the exported features of the composing Statecharts.
regards entity specialisation as reflected by state specialisation. The idea which inheres most methodologies is therefore that a system can be best understood by first abstracting its static structure, essentially in terms of object and associations between the objects, subsequently adding dynamic knowledge to the skeleton of structural relationships, either in terms of events or messages depending on the abstraction level. This way behaviour is attached to a domain which is already structured, reflecting the naive view that the static and dynamic aspects of the domain are separated and do not influence each another.

The subtle point suggested by the paper is that dynamic aspects may act as a guide for finding the effective structure of the domain. Shaping parts and wholes around observed synchronisation yields behavioural abstractions which can be further composed, extended and understood. The high level behavioural abstractions resulting may in other words be reused as such or translated to object types, which in their turn realise software quality factors through improved self-containment and encapsulation.

References


Figure J. Extending the example by dropping balls from different nozzles.