Vrije Universiteit Brussel
Faculteit Wetenschappen

Incremental Design of Layered State Diagrams

Tom Mens Patrick Steyaert

Techreport vub-prog-tr-97-04

Programming Technology Lab (PROG)
Faculty of Sciences
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Incremental Design of Layered State Diagrams

Tom Mens
tommens@vub.ac.be
Department of Mathematics

Patrick Steyaert
prsteyae@vnet3.vub.ac.be
Department of Computer Science

Faculty of Sciences, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
http://progwww.vub.ac.be/

Abstract. *Nested state diagrams are a commonly accepted design notation for modelling complex software systems at different levels of detail. Besides nesting, other layering mechanisms are needed. However, incremental development of layered state diagrams raises the problem of change management. Changes made at high abstraction layers can have a serious impact on more concrete layers, as conflicts can be introduced that are propagated through all layers. To resolve these conflicts, the methodology of* reuse contracts *is applied. Carefully choosing a set of incremental modification operators and investigating their interactions allows us to categorise the conflicts. These conflicts can be detected semi-automatically, facilitating incremental design.*

1. Introduction

Modelling complex software systems is hard. On the one hand software models should be understandable and simple enough to allow intuitive reasoning about them. On the other hand they must be detailed enough to be realistic. So, in order to deal with the complex structures one is usually confronted with, it is essential to model the world at different levels of detail. This claim is even more valid for state diagrams, where the number of states and transitions tends to increase combinatorially as the system grows in size.

For the design of software systems, *layering* is a common solution to this problem. Rather than building a flat model, system components are described at different levels of abstraction. For state diagrams, nesting of states is such a layering mechanism. Nested states increase the amount of detail without unnecessarily cluttering the design at higher levels of abstraction. Nested state diagrams were introduced by Harel under the form of statecharts [5]. Variants of statecharts are widely used and accepted in object-oriented analysis and design methodologies like Booch [1], OMT [8], UML [2] and Syntropy [3], and in real-times methodologies like ROOM [10]. While nesting is important, it is just one of the many different ways to create new layers in a state diagram: a state diagram can be *extended* by adding entirely new behaviour, the behaviour of its states can be *refined*, or parts of the diagram can be *specialised* or *encapsulated*. These operators correspond to different intuitive ways in which a design can be incrementally modified, and just like nesting they should be explicitly supported.

Layering is not only crucial for mastering complexity. It is also needed when describing different versions of a particular system. To do so, the commonalities between the different versions are described in an abstract layer; the differences are described in separate concretisations. This has the advantage that differences and commonalities are made explicit. It allows reasoning about the common system in general, and about different versions in particular. This, however, raises the question of version and *change management*: What is the effect of changes made to the general system on particular versions? How do these different versions interact ? For example, changes made to an abstract layer usually have a large impact on the different versions that were derived from it. In some cases, this might be what is desired, in other situations it can lead to inconsistent behaviour that is hard to detect. In general, we need to be able to assess the impact of changes that propagate through the different layers of a model. This is essential because complex models constantly need to cope with change. Mechanisms are needed that help in detecting inconsistencies due to the propagation of these changes. Current methodologies and tools do not go beyond detection of purely syntactic conflicts.

This paper proposes a methodology that fully supports layering and change management for state diagrams. It is based on a variant of statecharts, enhanced with a range of incremental modification operators. We investigate how these operators interact, which allows us to categorise the conflicts that can occur when an abstract layer is modified, and to automatically pinpoint the places where *potential* problems can arise. Based on this, powerful tools can be developed that support the user during incremental design. The chosen approach corresponds to the general methodology of *reuse contracts* [12]. By applying this methodology to nested state diagrams, we show how incremental design of such diagrams can be facilitated.

2. Nested State Diagrams (NSDs)

2.1 Definition

To keep the presentation clear, we have chosen to present the definitions in this paper in a semi-formal way. We will use a notion of state diagrams that combines statecharts [5] and OMT state diagrams [8]. In order to reduce complexity we have imposed several restrictions on these diagrams. For example, we do not allow concurrrent substates or transitions between states at different nesting levels. The exact definition of our state diagrams is given below, while the semantics will be postponed until the following subsection.

A **nested state diagram** (**NSD**) is a directed graph consisting of states and transitions between those states. Each transition has two associated states: its *source state* and its *target state*. States as well as transitions are labelled. Exceptions to this are *initial states*, *terminal states* and *automatic transitions* that do not have a label. All states and transitions should satisfy the following conditions:

(1) Labelled states can enclose a nested state diagram. The enclosing state is called a *superstate*; the enclosed states are called *substates*.

(2) All states at the same abstraction level have a different label. There is exactly one initial and terminal state at each abstraction level.

(3) Transitions are only allowed between states at the same abstraction level. Transitions with the same source state have different labels. There can also be at most one automatic transition from each source state.

(4) There are no transitions with the terminal state as source state. There are no transitions with the initial state as target state. There is at most one transition with the initial state as source state, and this transition is always automatic.

(5) If there is a transition with a terminal substate as target state, there must also be an automatic transition with its superstate as source state.

In all the subsequent examples the notation of Figure 2 will be adopted. This notation is similar to the one used in most methodologies.

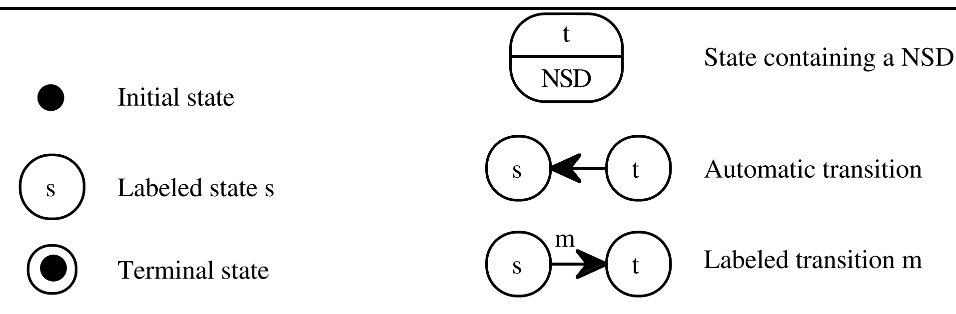

Figure 2: graphical notation of NSDs

2.2 Semantics

To avoid confusion with models that adopt a different semantics than ours, we specify here which meaning should be attached to the constructs defined in the previous section. Our semantics is similar to, but more restricted than, the one proposed in [6].

- *Labelled transitions* are transitions with an associated event. Whenever the event occurs, the transition immediately fires. *Automatic transitions* are transitions that fire automatically once the activity in the current state is finished. Automatic transitions do not have an associated event. Hence they cannot interrupt ongoing activities. They only occur after the activity in a state is finished (i.e. when the terminal substate is reached). This is an essential difference with labelled transitions.

- A *terminal state* is a state in which the execution stops. Since this may lead to unforeseen termination of the system, condition (5) in the definition of NSD is needed to recover from this situation. An automatic transition starting from the superstate can be used to resume execution at a higher abstraction level.

- Whenever a transition leads to a superstate, its unique initial substate will be visited first. In other words, all *incoming transitions to a superstate* are redirected to the unique initial substate.[1]
- An *outgoing automatic transition from a superstate* intuitively corresponds to an automatic transition from the terminal substate. An *outgoing labelled transition from a superstate* corresponds to an outgoing transition from *each* labelled substate. However, when a substate already has an outgoing transition with the same label, the transition from the substate has precedence over the transition from the superstate. This so-called *overriding of transitions* is a direct result of the nested scoping mechanism.

2.3 Digital Clock Example

As an example, consider the NSD of Figure 3 representing the behaviour of a digital clock.[2] Basically, a digital clock has two buttons: a **set** button and a **mode/adjust** button. When the **set** button is pressed, the event S occurs. Pushing the **mode/adjust** button gives rise to event M. Essentially the behaviour of the clock is split up into Display behaviour and Set behaviour. By consecutively pushing the **mode/adjust** button the owner of the clock can switch between the different display modes Time (displaying hour and minutes in HH:MM format), Date (displaying month and day in MM:DD format) and Second (displaying the seconds in :SS format). Alternatively, by pushing the **set** button the user can traverse the different set modes TimeSet (subdivided in an Hour set and Minute set state) and DateSet (subdivided in a Month set and Day set state). In all specific set states the **mode/adjust** button can be pushed to increment the current time or date. In the Second display state, pushing the **set** button resets the seconds (to simulate stopwatch behaviour). Finally, the Display automatically changes from Date to Time after a time-out of say 3 seconds during which no buttons have been pressed.

Remark that the behaviour inside Display is modelled as an infinite loop. Nevertheless, the labelled transition S from Display to Set allows us to interrupt each Display substate (except Second in which S is overridden) to commence set behaviour. Conversely, the automatic transition from Set to Display indicates that the TimeSet and DateSet activities must be completely finished before we can return to the display behaviour.

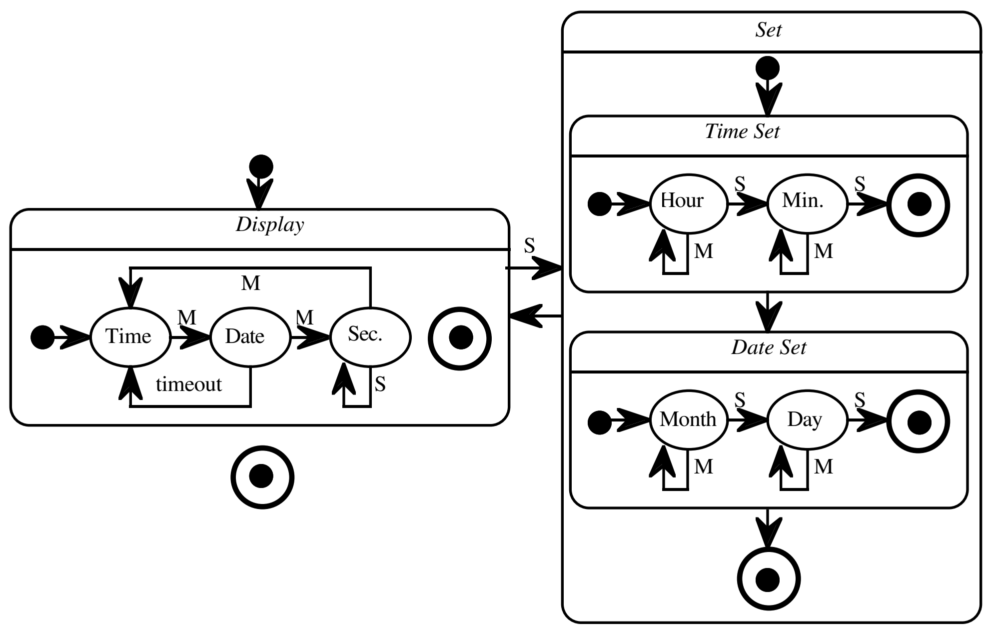

Figure 3: Digital Clock Behaviour

---

[1] Other semantics are possible, for example by making use of the so-called *history state* of [5]. This design choice however is irrelevant to the discussion in the paper. This paper does not try to capture all possible semantics and corresponding conflicts, but rather gives a *methodology* for facilitating incremental design in the context of state diagrams.

[2] This example and variants thereof will be used throughout the whole paper.

3. Layering Versus Change

The example of the previous section models a clock with minimal functionality only. We can identify two fundamentally different ways of adapting it. First, enhancements can be achieved in an incremental fashion by means of layering. Examples of this are an alarm clock, a stopwatch, a chime, switching between different time zones, etc. A second kind of changes alter the base functionality, and need to be made to the model directly: adding additional buttons, removing time-outs, etc. These changes are typically the result of an *iterative* design process. Obviously, they can have a (negative) effect on the behaviour of previously added layers.

To incrementally add enhancements, nesting alone does not suffice. For example, it might also require insertion of new states between existing states. This is the case when adding the functionality of a stopwatch: an extra Stopwatch state must be inserted between the Second and Time substate of Display. In the next section we define different modification operators for adding extra layers to a state diagram: extension, refinement, specialisation (nesting), and encapsulation. While not the only operators conceivable, they correspond to typical ways to incrementally modify a state diagram.

As depicted in Figure 4, the modification operators can be used for adding layers as well as for iterating over a base model. The interaction between the operator that is used to change an abstract base layer and the operators that were used for deriving concrete layers determine the kinds of conflicts that can arise in these concrete layers. In section 5 we will investigate the interaction between the different modification operators, and we will show how possible conflicts can be automatically detected.

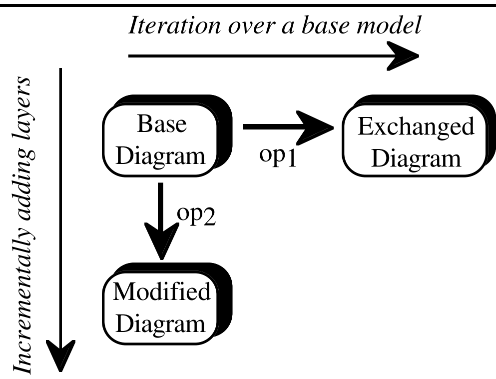

Figure 4: Modifying a Layered Design

4. Modification Operators for NSDs

In this section we propose a set of modification operators that coincide with the typical ways a layered design can be incrementally modified. *Extension* adds new states to the NSD without affecting the behaviour of existing states, while *refinement* refines the behaviour of existing states. A *specialisation* operator enables creation of nested state diagrams. Finally, an *encapsulation* operator - needed to cope with the combinatorial explosion of states - encapsulates a set of states into a new superstate. For each of these operators, a definition and a description of the prerequisites are given below. The prerequisites determine if an operator is applicable to a given diagram D_1. The definition itself provides an algorithm to construct the modified diagram D_2.

Since it is sometimes useful or even necessary to breach part of the design during incremental development, we will also need the inverse operators of the ones proposed above. *Generalisation* is the inverse of specialisation, and is used to remove internal details. *Cancellation* is the inverse of extension, and is mainly used to remove transitions. *Coarsening* is the inverse of refinement. It deletes states (and their outgoing transitions) from the diagram. *Decapsulation* is the inverse of encapsulation. While encapsulation groups related states together inside a superstate, decapsulation removes this superstate so that the substates are again directly accessible. Due to space limitations, we will not be able to discuss these inverse operators in detail.

4.1 Extension

Extension provides a mechanism to insert new reachable states into a NSD without affecting the behaviour of any of the old states, i.e. the set of events that are understood and the number of transitions that arrive remain unaltered. This makes extension a fairly safe operator that does not cause many conflicts with the more concrete layers. An example of the extension mechanism is shown in Figure 5, where the transition M from Date to Time is replaced by a sequence of two transitions M with intermediary state Second.

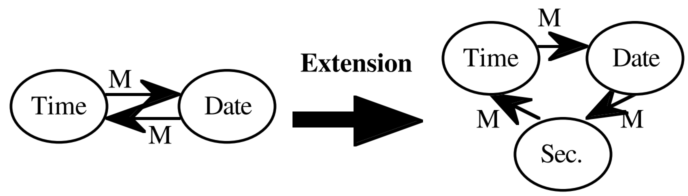

Figure 5: Extension of a NSD

D_1 is **extendible** with (t_{old},t_1,s,t_2) if:

(1) t_{old} is a transition of D_1 with the same source state and label as transition t_1, and the same target state as transition t_2

(2) s is a new labelled state at the same abstraction level as the source and target states of t_{old}, but with a label different from all other states at the same abstraction level

(3) s is the target state of t_1 and the source state of t_2

D_2 is an **extension** of D_1 with (t_{old},t_1,s,t_2) if

(1) D_1 is extendible with (t_{old},t_1,s,t_2)

(2) D_2 is obtained from D_1 by successively removing the transition t_{old} from D_1 and adding the transitions t_1 and t_2 with intermediary state s.

A composition of extensions is also called an extension.

4.2 Refinement

The extension operator does not suffice when the behaviour of some states needs to be refined with extra events. This is the case when new states are introduced that are related to already existing states. For these situations we will need a *refinement* operator. Clearly this operator is likely to give more conflicts in the lower layers than the extension operator.

In Figure 6 the NSD on the left is refined by adding a new state DateSet. Since DateSet is related to DateDisplay in the same way as TimeSet is related to TimeDisplay, the behaviour of DateDisplay needs to be refined with an extra S transition leading to DateSet, that also contains a transition S leading back to DateDisplay.

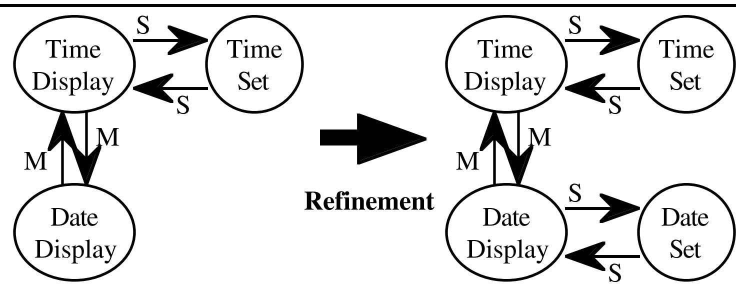

Figure 6: Refinement of a NSD

D_1 is **refinable with** (S,T) if

(1) S is a possibly empty set of new labelled states at the same abstraction level with different labels than the states of D_1 at this abstraction level

(2) T is a set of transitions (possibly automatic) with source and target states that belong to D_1 or S and that are at the same abstraction level. T contains no transitions with the same source state and label (possibly automatic) as other transitions in D_1 or T.

(3) Each state in S is the target state of a transition in T that has a state of D_1 as source state.

D_2 is a **refinement** of D_1 with (S,T) if

(1) D_1 is refinable with (S,T)

(2) D_2 is obtained from D_1 by first adding the states of S and then adding the transitions of T.

A composition of refinements is also called a refinement.

4.3 Specialisation

Some states in a NSD can be expanded to an entire new NSD. In OMT [8] this operator is referred to as state generalisation, while [4] and [7] call it decomposition. We think that the term *specialisation* is more appropriate, and will reserve the term *generalisation* for the inverse operator.

Usually, when performing an incremental design, we start with states that correspond to abstract activities. Specialisation corresponds to specifying some activities into more detail. An example of specialisation of the Set state is presented in Figure 7. Note that specialisation on its own is a pretty useless operator. In practice, specialisation will always be accompanied by a refinement or extension, in order to fill in the details of the specialised state. This process can go on as long as necessary, since the substates themselves can also be specialised and filled in.

D_1 is **specialisable** with s if

(1) s is a labelled state in D_1 that does not (yet) contain any substates

D_2 is a **specialisation** of D_1 with s if

(1) D_1 is specialisable with s

(2) D_2 is obtained from D_1 by adding inside s an initial and terminal substate.

A composition of specialisations is also called a specialisation.

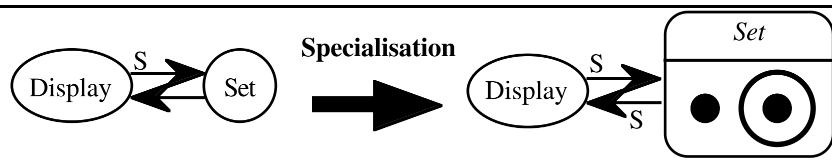

Figure 7: Specialisation of a NSD

4.4 Encapsulation

Encapsulation is used to restructure the diagram in a way that makes it (hopefully) more manageable and understandable. Encapsulation groups a set of existing states together inside a new superstate. Afterwards the encapsulated states are only accessible via their enclosing superstate. This operator is very useful, since state diagrams have a tendency to grow very rapidly in the number of states and transitions.

Before giving the definition of encapsulation, consider the example of Figure 8 where two encapsulations are performed. TimeDisplay and DateDisplay are encapsulated inside a Display state, while TimeSet and DateSet are encapsulated inside a Set state. Because of this encapsulation, several transitions need to be moved, in order to satisfy the well-formedness of the new diagram and the semantics of the original diagram:

- First of all, the two transitions with label S from TimeDisplay and DateDisplay to TimeSet are replaced by only one transition with label S from the Display superstate to the Set superstate. By designating TimeSet as the first substate following the initial substate of Set (TimeSet is called the *post-initial substate* of Set), the net effect of the transitions will remain the same.
- Secondly, the transition S from DateSet to TimeDisplay is replaced by a transition S from DateSet to the terminal substate of Set, followed by an automatic transition from Set to Display. DateSet is called the *pre-terminal substate* of Set.

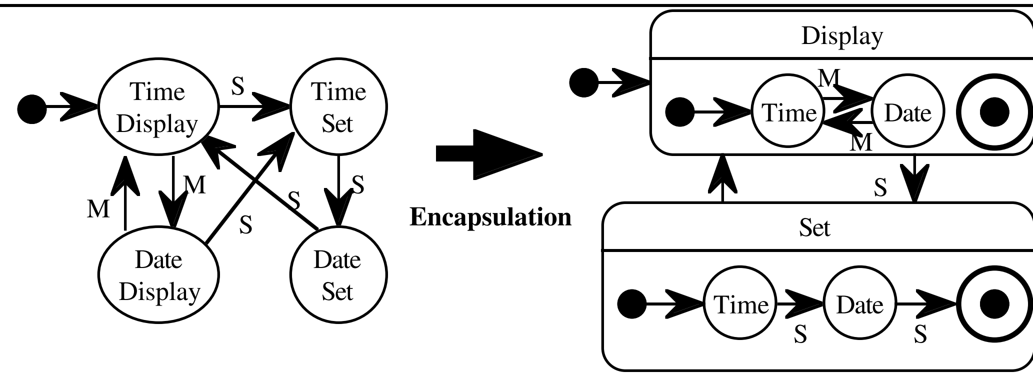

Figure 8: Encapsulation of a NSD

Because of these delicate interactions between different abstraction levels, the definition of encapsulation is fairly complicated.

D_1 is **encapsulatable** with (s_{new},s_i,s_t,S) if

(1) S is a set of labelled states of D_1 that all appear at the same abstraction level. s_i and s_t are two states in S that are called the *post-initial state* and *pre-terminal state* respectively.

(2) s_{new} is a new labelled state at the same abstraction level as the states of S, but with a label different from other states in D_1 (excluding the states of S) at the same abstraction level.

(3) All transitions with source state outside S and target state inside S should have s_i as target state.

(4) There can be one transition with s_t as source state and a state outside S as target state. All other transitions from inside S to outside S must satisfy (5).

(5) If there is a transition —different from the one in (4)— with source state inside S and target state outside S, there must be transitions with the same label from all states inside S to the same target state.

D_2 is an **encapsulation** of D_1 with (s_{new},s_i,s_t,S) if

(1) D_1 is encapsulatable with (s_{new},s_i,s_t,S)

(2) D_2 is obtained from D_1 by successively

a) Removing all the states of S from D_1

b) Adding state s_{new} to D_1

c) Adding an initial state, a terminal state and all the states of S inside s_{new}

d) Adding an automatic transition from the initial substate to s_i.

e) Replacing transitions in D_1 with s_i as target state by transitions with superstate s_{new} as target state.

f) Replacing all transitions with the same label in D_1 from each element of S to a target state t outside S by a single transition with the same label from s_{new} to t.

g) Replacing the transition of (4) from s_t to a state t outside S by a transition with the same label from s_t to the terminal substate of s_{new}, followed by an automatic transition from s_{new} to t.

A composition of encapsulations is also called an encapsulation.

Encapsulation and specialisation can be regarded more or less as dual operators. While encapsulation adds a new abstraction layer around existing states, specialisation makes an abstract state more concrete by adding new states (and transitions) inside it. For reasons of orthogonality, specialisation and encapsulation are the only operators that can add new initial and terminal states. Insertion of initial and terminal states is excluded from the definition of extension and refinement.

5. Conflicts Between the Operators

As already mentioned in section 3, detection of conflicts during incremental modification of a layered design requires a careful investigation of the interaction between the different modification operators. During this investigation, we will adopt the terminology of Figure 4.

When discussing the conflicts between the modification operators, we will distinguish two different categories. Syntactic conflicts indicate interaction problems between the different operators on a syntactic level; a typical example is name collisions. Semantic conflicts are more subtle. If they go unnoticed, the behaviour of the system might be changed inadvertently. While syntactic conflicts can be detected automatically, this is not always the case for the semantic ones. Nevertheless, a careful analysis of the interaction between the operators will enable us to automatically pinpoint the places where *potential* semantic problems might arise.

5.1 Syntactic Conflicts

5.1.1 State and Transition Collisions

A common source of syntactic conflicts are state and transition collisions. They can occur with any combination of extension and refinement, whenever a state or transition with the same label is added in the exchanged as well as in the derived diagram. Note however that there is an essential difference between these two kinds of collisions. While state collisions can usually be solved by renaming one or both state labels, this is not the case for transition collisions, as the transition label is essential in the design of the diagram!

Assume that in the modified diagram as well as in the exchanged diagram a transition with the same label and source state is added.[3] If these transitions have the same target state in both diagrams, everything is all right. If both transitions have different target states, this leads to a *transition collision*. An example of this is illustrated in Figure 10, where there is a transition S from DateSet to DateDisplay in the exchanged diagram, and a transition S from DateSet to TimeDisplay in the modified diagram. In order to solve this conflict, we need to ask the user for help, since it is impossible to automatically derive which one of both transitions is more appropriate.

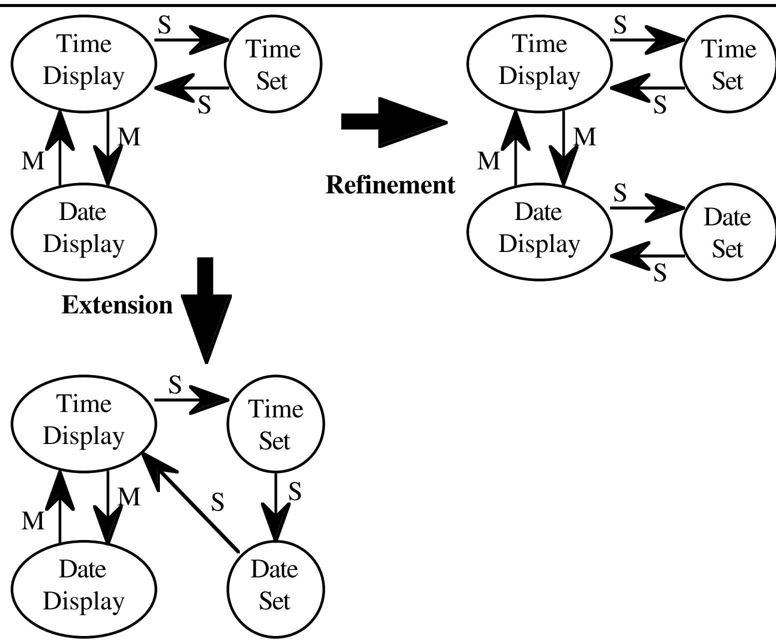

Figure 10: Unintended transition collision

5.1.2 Encapsulation Conflicts

Encapsulation is an operator that conflicts with every other operator due to its complexity. To see this, consider the example of Figure 11. The exchanged diagram is obtained by encapsulating TimeDisplay and DateDisplay inside a Display superstate. The modified diagram is obtained by extending the original diagram with a new DateSet state. Even in this simple situation, a lot of conflicts can be detected.

---

[3] Also in the case where both transitions are automatic a conflict will occur.

First of all, we have an *undecidable encapsulation conflict*. Indeed, if we want the modified diagram to be an extension of the exchanged diagram, we need to decide whether or not the newly introduced DateSet state should be encapsulated inside the Display superstate. Although it is clear that in this case DateSet should be left outside the Display state, such a decision cannot be made automatically.

Once we have decided to leave DateSet outside the Display state, other conflicts will emerge. One of these is the so-called *ungroupable incoming transition* conflict. The transition S from DateSet to DateDisplay in the modified diagram cannot be applied to the exchanged diagram since this would breach the encapsulation imposed by the Display state. Changing the target state of the transition from DateDisplay to Display is also no solution, since this would lead to the post-initial substate TimeDisplay instead of DateDisplay. If we observe more closely, the ungroupable incoming transition conflict occurs because DateDisplay is not post-initial inside the Display state.

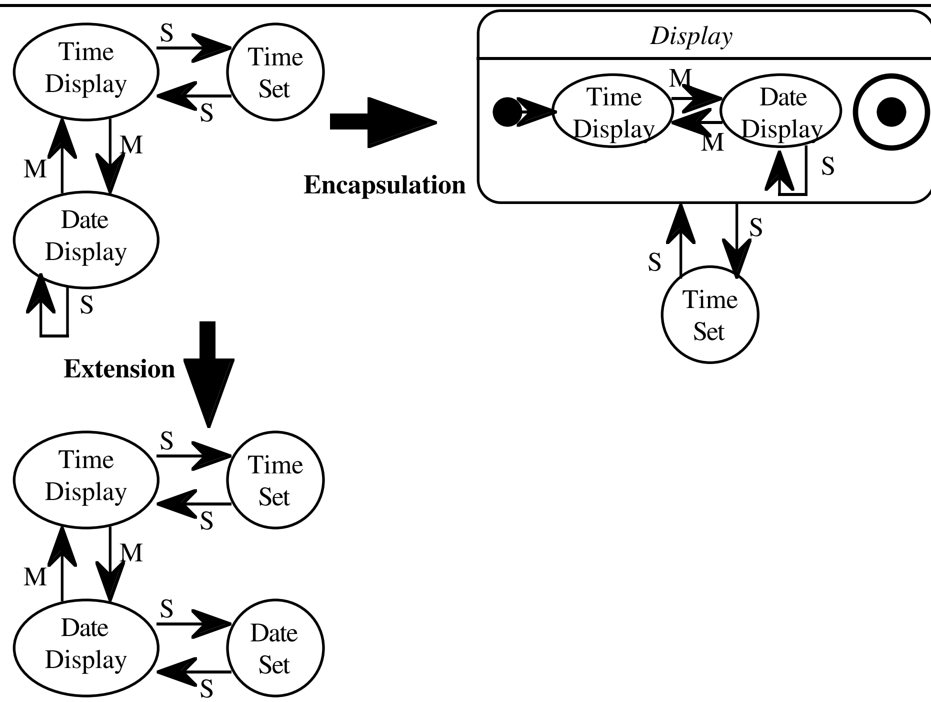

Figure 11: Undecidable encapsulation, Ungroupable transition

A third problem that can be detected in Figure 11 is the *ungroupable outgoing transition* conflict, which is the dual of the previous conflict. For example, the transition S from DateDisplay to DateSet in the modified diagram leads to a conflict since DateDisplay is encapsulated inside superstate Display, while DateSet remains outside Display. For this reason the transition S from DateDisplay to DateSet cannot be applied anymore as it would breach the encapsulation imposed by the Display state.

Note that the three previous conflicts can also occur in the case of refinement versus encapsulation.

A final and trivial conflict concerning encapsulation occurs whenever two non-disjoint sets of states are encapsulated in the modified and exchanged diagram respectively. This conflict of *undecidable encapsulation* indicates that two overlapping encapsulations have been made.

5.1.3 Summary

In Table 1, an overview of the various syntactic conflicts is presented. Only the upper left triangle of the table is filled in since the table is symmetric. We can make several observations:

- Because they both have the ability to add states and transitions, extension and refinement give rise to the same syntactic conflicts.
- Whenever one of the operators is an encapsulation, no transition collisions can occur. This is logical, since the encapsulation operator only adds a new superstate to the diagram.

- Whenever one of the operators is specialisation, no collisions occur at all, since specialisation does not add any transitions or labelled states. The only thing a specialisation adds to a diagram is an initial state and a terminal state, and those cannot be added by any of the other operators.

Exchanged / Derived	Specialis.	Encapsulation	Extension	Refinement
Refinement	✔	State collision Ungroupable transition Undecidable encaps.	State collision Transition collision	State collision Transition collision
Extension	✔	State collision Ungroupable transition Undecidable encaps.	State collision Transition collision	
Encapsulation	✔	State collision Undecidable encaps.		
Specialisation	✔			

Table 1: Overview of the syntactic conflicts

5.2 Semantic Conflicts

5.2.1 Nesting Conflicts

When changes are made in different layers of the design, two important semantic conflicts can occur. When substates unintendedly become interruptable by a transition defined on a superstate, a *transition capture* occurs. *Transition overriding* can occur when a transition defined on a superstate becomes overridden by a transition on a substate. Both kinds of conflicts are possible when the modified/exchanged diagrams are obtained via refinement/extension, extension/refinement or extension/extension.

When extra substates are added inside an already specialised state in the modified diagram, and the exchanged diagram is obtained by adding transitions with this superstate as source state, we have a *transition capture*. All newly added substates in the modified diagram will become interruptable by the transition added in the exchanged diagram. An example is given in Figure 12. The Display state, containing Time and Date substates, is exchanged for a new diagram where all substates can be interrupted by a time-out event. When a Second substate is added in the modified diagram, it will also be interruptable by a time-out event. However, this was not the intention of the person that performed the extension. Indeed, most clocks do not allow the second display to be interruptable by a time-out, to be able to use the clock as a rudimentary stopwatch.

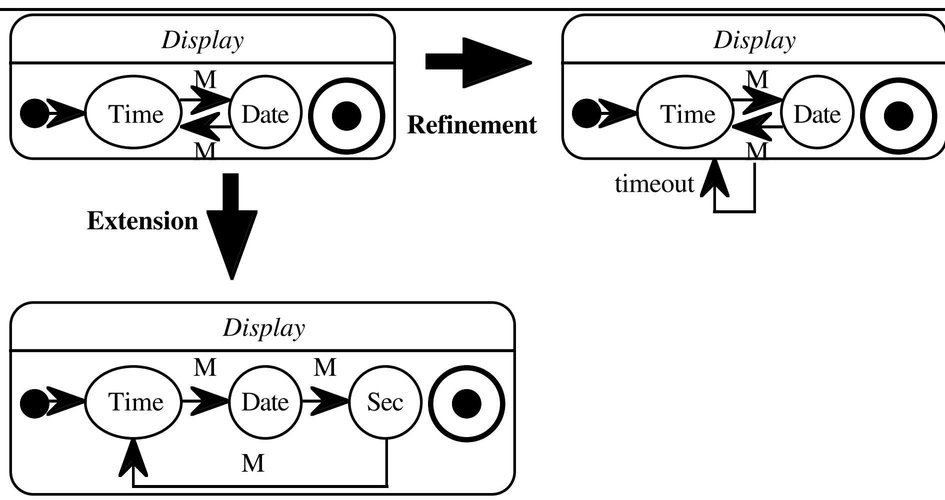

Figure 12: Transition capture

A *transition overriding* can occur when transitions with the same label are added at two different abstraction levels. A transition in the modified diagram might override a higher-level transition in the exchanged diagram. As an example, consider the base diagram of Figure 13 containing a Display state with Time, Date and

Second substates. The exchanged diagram is obtained by adding an extra TimeSet state that can be reached by following the transition S from Display. The modified diagram is also obtained through refinement by adding a transition S in the Second substate. Pressing the S button will reset the seconds. This is useful when adding stopwatch behaviour. The modified diagram is in conflict with the exchanged diagram as it becomes impossible to reach TimeSet from inside the Second substate, since the transition S on Second overrides the transition S on its superstate Display. Again it is impossible to determine automatically whether or not this conflict is intended.

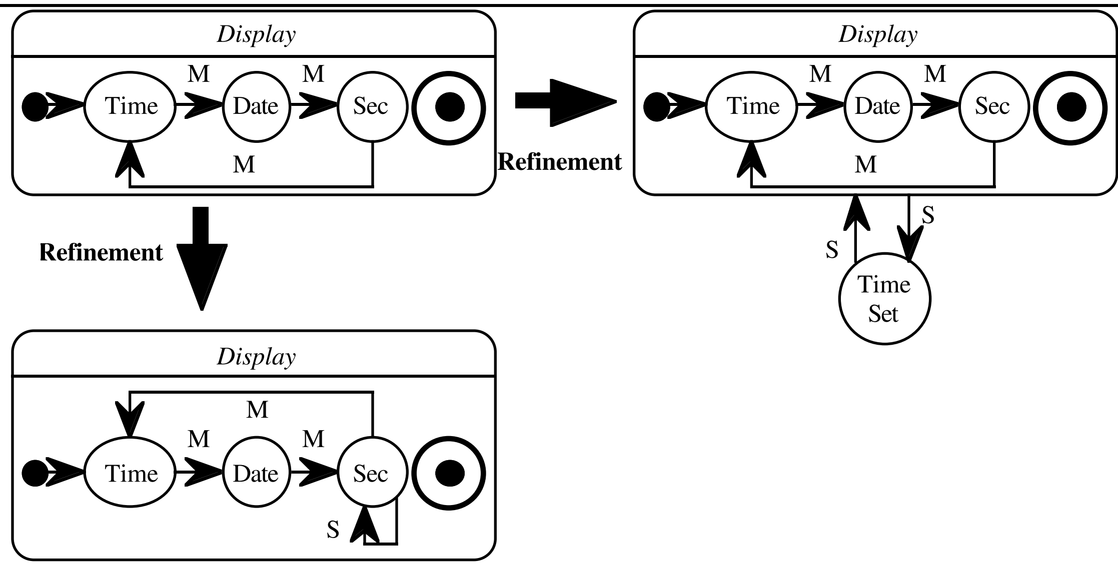

Figure 13: Transition overriding

5.2.2 Reachability Conflicts

Another kind of conflicts concerns reachability of states. By specialising a state, other states can become unreachable in certain situations. The opposite problem occurs when states are reached that should not be reachable.

Adding in the exchanged diagram an automatic transition with a source state that has not yet been specialised means one immediately arrives in the target state of the automatic transition. However, suppose that the source state is specialised in the modified diagram. In that case the target state of the automatic transition in the exchanged diagram becomes unreachable until a transition to the terminal substate of the specialised source state is added. This might never be the case if the substates of the specialised state are only intended to be interruptable by a specific event. We call this an *unreachability conflict*. As an example, consider the base diagram of Figure 14 containing only abstract states Display and Set. In the exchanged diagram automatic transitions between both states are added. In the modified diagram, Display is specialised in such a way that the terminal substate is never reached. Then the automatic transition from Display to Set in the exchanged diagram will never be fired.

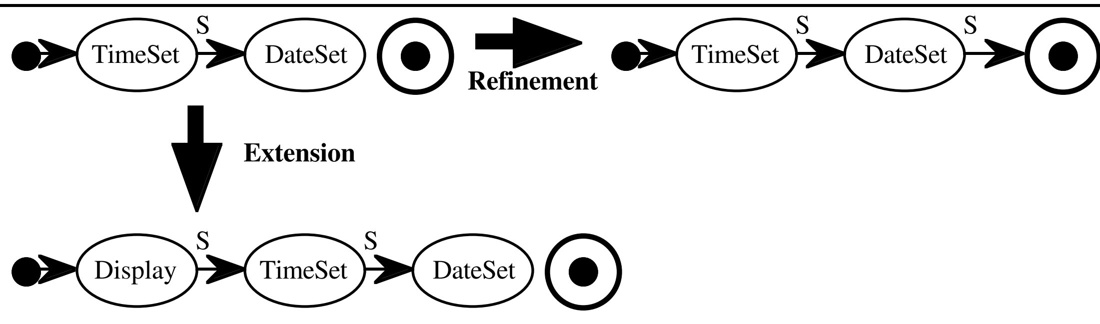

Figure 14: Unreachability conflict

An *unintended reachability conflict* can occur when states that should not be reachable suddenly become reached. The essential problem here is that state diagrams do not provide sufficient means to express design information to prohibit some states from being reached. For example, in Figure 15 we start from a base diagram describing only set behaviour. In the exchanged diagram, a transition S from DateSet to the terminal state is added. In the modified diagram a new Display state is inserted before TimeSet to include display behaviour. (Further modifications of the modified diagram will add a transition S from DateSet back to Display in order for the clock to work correctly.) Then there clearly is a conflict between the modified and exchanged diagram. Inserting Display in the exchanged diagram would not lead to useful behaviour since the clock would simply stop running when the event S would be issued in DateSet! In other words, the terminal state should not be reachable from DateSet.

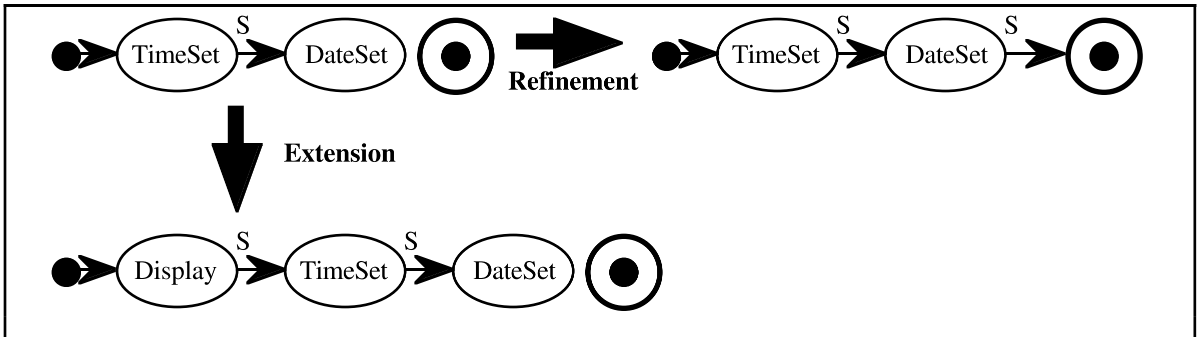

Figure 15: Unintended reachability conflict

5.2.3 Summary

In Table 2 an overview of the various semantic conflicts is presented. Unlike the syntactic conflicts, where specialisation gave no problems, encapsulation gives no semantic conflicts. As predicted, extension is a safe operator that does not cause many conflicts with the more concrete layers. Refinement on the other hand is a dangerous operator, since it can change the behaviour of existing states in an unforeseen way, leading to conflicts in lower layers.

Exchanged Derived	Extension	Specialisation	Refinement	Encaps.
Encapsulation	✔	✔	✔	✔
Refinement	Unintended reachability Transition overriding Transition capture	Unreachability	Unintended reachability Transition overriding Transition capture	
Specialisation	✔	✔		
Extension	✔			

Table 2: Overview of the semantic conflicts

6. Related Work

The approach taken in this paper essentially corresponds to the methodology of reuse contracts presented in [12]. Apart from the fact that we have applied this methodology to state diagrams instead of object-oriented class hierarchies, there are some other differences. First of all, since we allowed to deal with nested states, we needed to provide operators like specialisation and encapsulation to modify the nesting structure. Secondly, the conflicts have been split up into syntactic and semantic conflicts. Finally, because of the plethora of different ways to implement a given state diagram, in this paper we did not discuss *concretisation* operators that add implementation details to a state diagram. However, if we agree upon a fixed implementation strategy for state diagrams, it will be fairly straightforward to define a *state concretisation* and *transition concretisation* operator, and to investigate their interaction with the other operators.

There have been several proposals for incremental modification mechanisms for state diagrams. Besides the fact that the modification operators proposed there differ from ours, none of the approaches investigate the interaction between the operators and the resulting conflicts. Hence it becomes very difficult to assess the impact of changes. Below we will briefly discuss the different proposals.

- [9] describes how state machines can be reused via object-oriented techniques such as subclassing, composition, delegation and genericity.
- Like in our approach, [7] presents a technique to incrementally construct a state machine for a class from the state machines of its base classes. The *strict inheritance assumption* is adopted, implying that states or transitions cannot be deleted in a child class, and new states can only be added as substates of existing ones. We disagree with this assumption, as it is sometimes useful or even necessary to breach part of the design during incremental development.
- In [6] an inheritance mechanism for statecharts is defined, also adopting the strict inheritance assumption. OMT [8] discusses a quite different inheritance model for statecharts, where substates are viewed as "subclasses" of their containing states.
- ROOMcharts [11] offer a variation of the statechart formalism that integrates object oriented features. Like in our approach, the strict inheritance assumption is rejected, and concurrent states are omitted from the formalism.
- In ObjChart [4], the object-oriented counterpart of statecharts, system refinement is only done by *object decomposition* (which is similar to our specialisation operator). While the authors of ObjChart believe that "*object decomposition* must be the predominant (and perhaps the sole) system refinement discipline", we are convinced that operators like extension, refinement, and encapsulation are at least as important.

7. Conclusion and Future Work

In this paper we applied the methodology of reuse contracts to facilitate incremental design of nested state diagrams. To cope with the complexity of software systems, several new layering operators besides nesting were added to state diagrams. To be able to assess and manage the impact of changes in such a layered design, we made a categorisation of the conflicts between the different layering operators. This categorisation allows us to semi-automatically detect these conflicts, and to develop powerful tools to assist and support the user during incremental design.

Despite all these results there is still a lot of work to do. Various enhancements to the NSDs presented in this paper are conceivable:

- Like in Statecharts and OMT we could allow for transitions that cross abstraction levels.
- Transitions could also be enhanced to include features like actions, activities and conditions as can be found in most methodologies. *Conditions* can be used as guards on transitions, only allowing a transition to fire if a certain condition is true. These conditions can even be subdivided into *pre*- and *postconditions*. When a transition fires, a certain *action* can be performed. We can also have *entry* and *exit actions* that occur when leaving the source state or entering the target state of the firing transition. Finally, *activities* can be associated to states.
- Based on the work of [5], *history states* could be used as an alternative to initial states, and *concurrent states* could be introduced to deal with concurrency.
- A final limitation is that we excluded nondeterminism from our model, since transitions with the same label cannot have the same source state.[4]

One has to pay attention when enhancing the definition of state diagrams with any of these features, since this will have a significant impact on the operators and conflicts that can occur. However, we are confident that most of the enhancements mentioned above can be added to our model in a straightforward fashion.

Although we have presented a powerful set of new layering operators in this paper, two issues should be examined further. First of all we should look at the conflicts that are introduced by the inverse operators. Secondly, we need to investigate other possible operators. A *rename* operator is necessary to rename states in the case of a state collision. A *promote* (and inverse *demote*) operator is useful to move existing states to a higher abstraction level. Operators that work on transitions rather than states are also useful. For example we could define operators like *transition specialisation* and *transition factorisation*.

While this paper only looked at incremental design by means of state diagrams, state diagrams are usually only one particular aspect of a methodology. They represent a behavioural or dynamic view of the system, describing how the system changes in time. For a structural or static view of the system, other formalisms like entity relationship diagrams and class diagrams are used. We are currently trying to apply the ideas of this paper to those formalisms as well.

---

[4] In fact, allowing more than one transition with the same label and source state does not necessarily imply nondeterminism. Only if the conditions on which the transitions can be fired are not mutually exclusive, nondeterminism can occur.

Another problem with describing the behaviour of a system through state diagrams is that usually separate diagrams are created for each class. This makes it difficult to visualise the behaviour of the whole system from a number of diagrams of individual classes. For this reason, interaction-based and event-based modelling techniques were invented. They enable the modeller to show how classes cooperate in a real system. An example is O-charts [6], a hierarchical OMT-like language for describing the structure of classes and their inter-relationships. The results presented in this paper can probably be translated to event-based diagrams as well.

8. Acknowledgements

We are deeply indebted to Carine Lucas, Kim Mens, Bart Wydaeghe and Kurt Verschaeve for the interesting discussions and useful suggestions about this topic. Also many thanks go to Marc Van Limberghen and Koen De Hondt for proof-reading the paper.

9. References

[1] Grady Booch: "Object-Oriented Analysis and Design With Applications", Addison-Wesley, 1994.

[2] Grady Booch & James Rumbaugh: "Unified Method for Object-Oriented Development", Rational Software Corporation, Version 0.91, 1996.

[3] S. Cook & J. Daniels: "Designing Object Systems: object-oriented modelling with Syntropy", Prentice Hall International, 1994.

[4] Dipayan Gangopadhyay & Subrata Mitra: "ObjChart: Tangible Specification of Reactive Object Behaviour", ECOOP '93 Proceedings, pp. 432-457, ed. Oscar Nierstrasz, LNCS 707, Springer-Verlag, 1993.

[5] David Harel: "On Visual Formalisms", Communications of the ACM, May 1988, Volume 31, Number 5, pp. 514-530, ACM Press, 1988.

[6] David Harel & Eran Gery: "Executable Object Modeling with Statecharts", ICSE '96 Proceedings, pp. 246-257, IEEE Press, 1996.

[7] John D. McGregor & Douglas M. Dyer: "A Note on Inheritance and State Machines", Software Engineering Notes, Vol. 18, No. 4, October 1993.

[8] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy & William Lorensen: "Object-Oriented Modelling and Design", Prentice Hall, 1991.

[9] Aamod Sane and Roy Campbell: "Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity", OOPSLA '95 Proceedings, pp. 17-32, ACM Press, 1995.

[10] B. Selic, G. Gullekson & P.T. Ward: "Real-Time Object-Oriented Modeling", John Wiley & Sons, New York, 1994.

[11] Bran Selic: "An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems", CHDL '93 Proceedings, 1993.

[12] Patrick Steyaert, Carine Lucas, Kim Mens & Theo D'Hondt: "Reuse Contracts: Managing the Evolution of Reusable Assets", OOPSLA '96 Proceedings, ACM Sigplan Notices Vol. 31, No. 10, pp. 268-285, ACM Press, 1996.