Specialization of Object Life Cycle Definitions

Jürgen Ebert
University of Koblenz
Dept. of Computer Science
Rheinau 1
D-56075 Koblenz
phone: ++49-261-9119-412
fax: ++49-261-9119-499
ebert@uni-koblenz.de

Gregor Engels
University of Paderborn
Dept. of Computer Science
Warburger Str. 100
D-33098 Paderborn
phone: ++49-5251-60-3337
fax: ++49-5251-60-3431
engels@uni-paderborn.de

December 1997

Abstract

Many object-oriented modeling approaches propose to use the concept of state transition diagrams to describe object behaviour. But most of them are very vague on which aspect of object behaviour should be described by them. In addition, the interrelation between the behaviour description of classes and those of their subclasses is rarely defined.

In this paper we motivate and formalize the distinction between two kinds of usages of state transition diagrams, called the observable and the invocable behaviour of objects. Based on this, we formally define compatibility conditions between behaviour description of classes and those of their subclasses depending on the kind of behaviour. We provide sufficient criteria for proving this compatibility and show that rules given by other authors are special cases thereof.

Keywords: object model, dynamic model, object life cycle, state transition diagram, specialization, inheritance

1 Introduction

An often mentioned characteristics of object-oriented modeling is the integrated description of structural as well as behavioural aspects of objects. In order to achieve this, current object-oriented development methods propose the usage of *class diagrams* for the description of the structural part and (variants of) *state transition diagrams* for the description of the behavioural part. Well-known examples of such methods are OMT ([RuBlPr 91]) or OOA ([ShlMel 92]), and UML ([BoRuJa 98]) follows this tradition.

Due to long experiences with the use of variants of entity-relationship-diagrams (the predecessors of class diagrams) in the area of conceptual modeling, concepts for the modeling of the *structural part* of an object are well understood. On the other hand, much less clarity exists about the meaning of the term *behaviour of objects* and how this can be modeled. Moreover the *integration* of the behaviour description and the structure description of objects is usually only rudimentarily explained in the literature.

The more object-oriented approaches are accepted by the software engineering community, the more there is a need for a formal semantics of object-oriented models. The formalization of semantics helps people to a deeper understanding of the underlying meaning of such models and gives a basis for the implementation of supporting tools. A lot of questions with respect to semantics of object-oriented models are still open, e.g. the interrelation between the behaviour descriptions of a superclass and its subclasses is only superficially described in the literature.

This observation caused several research groups to investigate this question in more detail. First published results can be grouped into constructive and descriptive approaches, respectively. Most of them deal with a state transition diagram (STDiagram) description of behaviour. *Constructive* approaches like [SaHaJu 94], [LopCos 93], and [McGDye 93] define rules how to modify the state transition diagram of a superclass to get a legal state transition diagram of a subclass. An analogous result has been published by [KapSch 94] in the context of Petri net like descriptions of object behaviour. *Descriptive* approaches like homomorphisms [EbeEng 94] or check conditions [SaHaJu 94] define restrictions which have to be fulfilled by a state transition diagram of a subclass with respect to the state transition diagram of a corresponding superclass.

With the assumption that a subclass inherits all methods from the superclass and may also have additional methods, most of these approaches follow the same intuition:

- ▷ A state transition diagram restricts the set of all sequences of method calls to those of the life cycles of an object, i.e. all *possible sequences* of method calls are described, which may be observed by clients of an object.
  The life cycle description of the subclass has to be compatible with the life cycle description of the superclass in the following sense: if we restrict a concrete life cycle of an object of a subclass to the methods defined also for objects of the superclass, we get a life cycle that is also allowed for objects of the superclass.

This approach reflects the idea that objects of a subclass should always also behave like objects of the superclass if they are viewed only according to the superclass description.

But, there are concrete examples in the literature (e.g. [McGDye 93] and [ShlMel 92]) or some ad-hoc intuitively drawn state transition diagrams, which do not fulfill the above mentioned conditions. Especially in modeling the behaviour of reactive systems one often wants to describe object behaviour following a slightly different intuition:

- ▷ The state transition diagram describes the really *executable sequences* of method calls, i.e. the sequences of method calls which may be invoked by clients of an object.
  Here, the life cycle description of the subclass has to be compatible with the life cycle description of the superclass in the following sense: the life cycle of a subclass may include additional method invocations, but all invocable sequences of method calls of a superclass have also to be invocable on objects of a subclass.

In this paper, we explain in detail these two different intuitions for modeling object behaviour and put them into a common framework. We describe how these two different intuitions are related to each other and how they interrelate with inheritance. The definitions and the formal discussions are based on flat state transition diagrams as the descriptional model, while for the ease of presentation a statechart-like notation is used in the examples.

This paper is an extension and generalization of the results presented previously [EbeEng 94]. While that paper was restricted to the first kind of modeling behaviour, we here discuss both aspects of behaviour specialization within a common framework. In section 2 we give concrete examples of the two different approaches to model object behaviour. Both approaches are formalized and discussed in section 3 and section 4. Section 5 relates our work to those of others and section 6 concludes the paper.

2 Meaning of Life Cycles

2.1 Observable Behaviour

One way to look at the state transition diagrams in object-oriented specifications is to interpret them as a description of all observable sequences of method calls, i.e. of sequences that *might occur* on objects of this class, though it is not guaranteed that all of these are actually executable or invocable with each instance of the class. The diagram is intended to be only a coarse description - ignoring more detailed semantic information - of the set of all method call sequences. E.g. an observable method call may not be invocable at a given point of time though being allowed by the diagram, since the actual value of the object does not fulfil certain additional preconditions of the method. Thus, the diagram describes an *upper bound* (in the sense of set inclusion) to the set of all possible call sequences. It is used to describe the set of all *observable* call sequences.

Given the interpretation of life cycle diagrams as a description of all observable sequences, we can immediately infer a condition on the compatibility of diagrams of super- and subclasses:

- ▷ each sequence of calls which is observable with respect to a subclass must result (under projection to the methods known) in an observable sequence of its corresponding superclass, since a subclass object must also behave as if it were an object of its superclass. Thus, if a subclass object reacts to a method call m (where m is also known to the superclass) this possible reaction must also be reflected in the superclass diagram.

Thus, if $OS(C)$ is the set of all observable sequences of method calls to a class C, we have for all superclasses C' of C:[1]

$$(OS(C) \upharpoonright M') \subseteq OS(C'),$$

[1] If s is a sequence over X and V is a subset of X, then $s \upharpoonright V$ is the sequence which contains just those elements of s which are members of V, in the same order as in s ([Spiv 92])

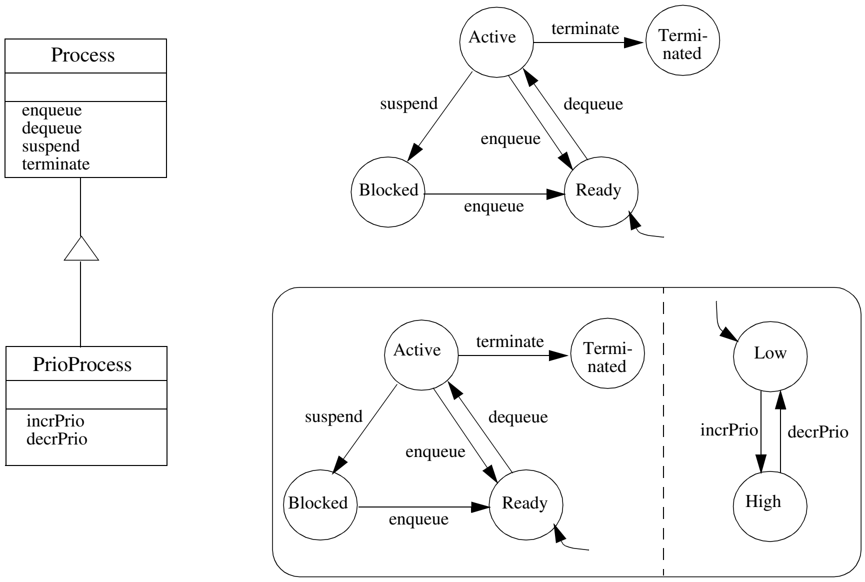


Figure 1: STDiagram of a subclass constructed by parallel extension

where M' is the set of method identifiers of class C', i.e. if we filter out all identifiers of methods that do not belong to C', we get a sequence which is observable with C'.

Example 1. As an example, we discuss the definition of a class *Process* as it is used by the scheduling components of operating systems and a corresponding subclass *PrioProcess* which adds the property of being in one of two priority levels to a process. Figure 1 gives the class definition as well as the behaviour description of these two classes.

As we are only interested in object life cycles in this paper, we omit the definition of attributes. Figure 1 shows that the behaviour description of the subclass *PrioProcess* is constructed by a parallel extension of the superclass state transition diagram. (We use Harel's statechart notation ([Har 87]) to yield a better readable representation than classical state transition diagrams. The set of states of the parallel extension is the cartesian product of the original sets (cf. [Ebert 93]).)

An observable (partial) call sequence, as defined by the state transition diagram of *PrioProcess*, is for instance:

... dequeue incrPrio suspend enqueue decrPrio dequeue ...
A restriction of this life cycle to the methods which are already defined within the superclass *Process* yields a life cycle which is allowed by the state transition diagram of *Process*. □

Example 2. As a second example, we give a subclass definition *LoggingProcess* in figure 2, where the corresponding state transition diagram is constructed by refining a state of the state transition diagram of *Process*. Again, the restriction of any life cycle which is observable for an object of *LoggingProcess* yields a life cycle of a *Process*. □

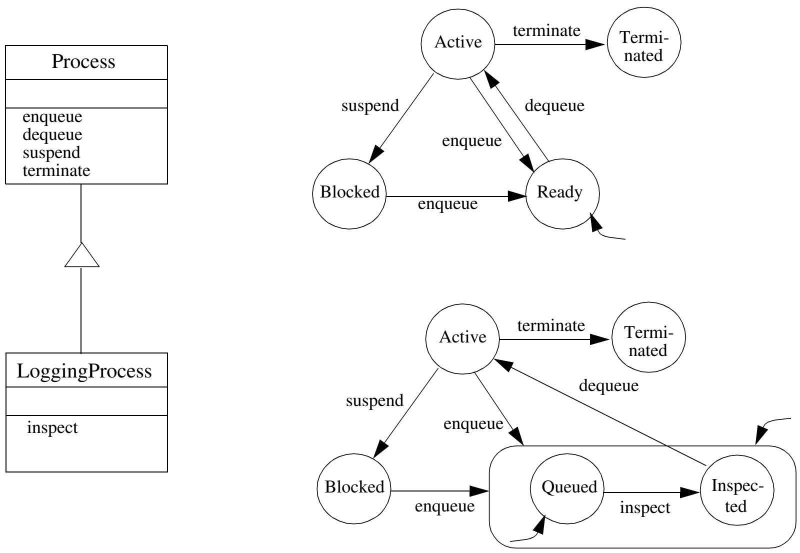


Figure 2: STDiagram of a subclass constructed by state refinement

In this example, the life cycle of a *LoggingProcess* is more "restricted" than that for a usual *Process*. To be a *LoggingProcess* means that it has to fulfill additional prerequisites ("to be inspected"), before it is allowed to be dequeued. Therefore, there exist invocable life cycles of a *Process* that are not possible for a *LoggingProcess*. This leads to the second kind of interpretation of state transition diagrams, the invocability approach.

2.2 Invocable Behaviour

Another way to interpret life cycle diagrams is to view them as a kind of usage contracts with their clients. Here, the diagram is intended to serve as a description of all *invocable* services that a client is able to use and where it is *guaranteed* that they are executable. Since a sequence of method calls uniquely determines a state in the diagram, the client now may arbitrarily choose between any of the methods that are given by the state's outgoing arcs. Then, it is guaranteed that all preconditions of the method are fulfilled, i.e. the method can be invoked and executed. In this context the diagram describes a *lower bound* (in the sense of set inclusion) of the set of all possible sequences and is used to describe a set of *invocable* call sequences.

Given the interpretation of life cycle diagrams as descriptions of invocable sequences of method calls we get a different condition on the compatibility of diagrams of super- and subclasses:

- ▷ each sequence which is invocable (or executable) with respect to a given superclass must also be invocable in all of its subclasses. Since each object which belongs to a subclass can be seen as belonging to the superclass, it must also have all the guaranteed methods as well as all invocable method sequences of the superclass.

Thus, if $IS(C)$ is the set of all invocable sequences of method calls to a class C and if, again, C is a subclass of C' we have

$$IS(C') \subseteq IS(C)$$

Example. As an example, we discuss the definition of a class *Tv* (television) and of a corresponding subclass *TvRemote* (television with remote control). Figure 3 gives the class definition as well as the behaviour description of these two classes. The class *Tv* provides only two primitive methods *up* and *down*, which allow to switch between 4 existing channels upwards or downwards, respectively, and a method *time* which shows the current time for some seconds on the screen. This method *time* may only be invoked at channel 4. The class *TvRemote* has in addition a remote control with four buttons *ch-i*, which enable the choice one of the four channels directly. The statechart of *TvRemote* describes this situation. It is a shorthand notation for a state transition diagram, where a state representing a channel *i* has incoming edges labelled by *ch-i* from all states.

Since the state transition diagram of *TvRemote* is an extension of the state transition diagram of *Tv*, all method sequences invocable on an object of class *Tv* are also invocable on an object of class *TvRemote*. But, in contrast to the above illustrated observable behaviour, the restriction of the following invocable method sequence on an object of class *TvRemote*

up ch-4 time up

to the methods already defined in class *Tv* yields the method sequence

up time up

which can not occur with an object of class *Tv*. Thus, with respect to observability *TvRemote* is not compatible to *Tv*. □

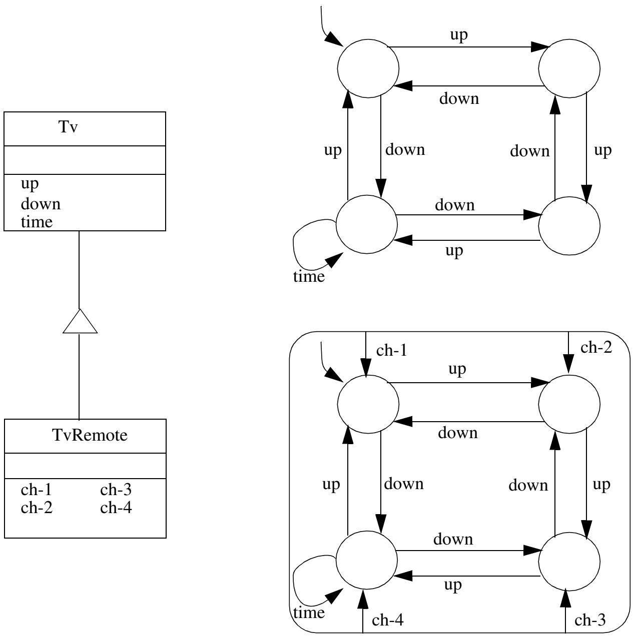


Figure 3: STDiagram of a subclass constructed by adding transitions

3 Formalization

In section 2 we have illustrated two different interpretations for using state transition diagrams as behaviour descriptions, the *observability* and the *invocability* approach. We have shown that depending on the chosen interpretation specific subset dependencies exist between the defined sequences of methods of a class and its subclass.

In order to guarantee these subset dependencies, the specified state transition diagrams have to fulfill certain constraints. It is the aim of this section to introduce a formal model for class definitions, state transition diagrams, and the interrelation between them. In section 4 we will apply this formalization to define constraints for the state transition diagrams and their interrelation for the observability and invocability approach, respectively.

The formalization goes along the lines of [EbeEng 94] and includes only as few concepts as necessary in order to describe the results of this paper. It is not intended to serve as a complete formalization of object oriented modeling. Thus, only method identifiers and state transition diagrams are included in the description.

At first, we shall define the object model including specialization and the dynamic model formally by corresponding mathematical constructs. This definition includes a formalization of the integration of both models by letting the dynamic model be state transition diagrams over the set of method identifiers (subsection 3.1). Then, the behaviour of objects is formalized. Based on the set of legal state sequences defined by the state transition diagram, the behaviour is defined as the set of method identifier sequences which correspond to legal state sequences (subsection 3.2). Finally, based on the existence of state transition diagram homomorphisms, two basic theorems are proved which imply the compatibility of behaviour under specialization (subsection 3.3).

The applications of these theorems to the concepts of observability and invocability will be discussed in more detail in section 4.

3.1 Structure

In the following we use a Z-like notation ([Spiv 92]) assuming some universes for identifiers, and states.

$$[\mathit{MID}, \mathit{OID}, \mathit{STATE}]$$

Here, *MID* stands for all possible method identifiers, *OID* is the stock of all possible object identifiers, and *STATE* is used for the states in the diagrams.

State transition diagrams (STDiagrams) are formalized as *finite automata* in the style of ([HopUll 79]). A state transition diagram consists of a finite set Q of states together with a start state q_0, an alphabet Σ of method identifiers, and a relation δ which describes possible state changes together with their possible inputs. Q stands for the vertices in the diagram and δ denotes its arcs. Here, the transition relation δ is non-deterministic and allows spontaneous ϵ-transitions.

STDiagram

$Q : \mathbb{F}\, STATE$	[states]
$\Sigma : \mathbb{F}\, MID$	[input alphabet]
$\delta : \mathbb{F}\,((Q \times (\Sigma \cup \{\epsilon\})) \times Q)$	[transition relation]
$q_0 : Q$	[start state]

A *non-deterministic* approach to modeling is used since specifications should be as general as possible. They should encapsulate as few design decisions as possible, leaving all details to the implementation. For the observability approach to state transitions diagrams, we have to use ϵ-transitions to model those cases where a method call to a specialization is not observable directly by the generalization but only seems to be a "spontaneous" change of state.

In order to formalize our results, a rudimentary definition of *classes* suffices, where classes are only described by their method identifiers and their behaviour - the latter given by a state transition diagram having the set of method identifiers as its input alphabet.

Class

$M : \mathbb{F}\, MID$	[method identifiers]
$STD : STDiagram$	[behaviour]
$STD.\Sigma = M$	

On the set of classes there is a *subclass relationship*, which gives rise to the existence of inheritance. In most object-oriented approaches it is assumed to be acyclic. We have no further semantic assumptions about inheritance, except that generally subclasses may have more methods than superclasses.

$$\begin{array}{|l}
(_\mathtt{isSubClassOf}_) : Class \leftrightarrow Class \\
\hline
isAcyclic(_\mathtt{isSubClassOf}_) \\
\forall C, C' : Class \mid C \mathtt{\ isSubClassOf\ } C' \bullet C'.M \subseteq C.M
\end{array}$$

There may exist a set of corresponding objects for each class description C at a given point of time[2]. Objects are represented by their respective object identifiers which stem from the common universe *OID*.

$$\begin{array}{|ll}
\multicolumn{2}{l}{\textit{Object}} \\
\hline
oid : OID & \text{[object identifier]} \\
\hline
\end{array}$$

Each existing object O belongs to exactly one class. It is said to be an *instance* of that class, i.e. there is a partial function `(_isInstanceOf_)` assigning class descriptions to objects.

$$\begin{array}{|l}
(_\mathtt{isInstanceOf}_) : Object \nrightarrow Class
\end{array}$$

The domain of `(_isInstanceOf_)` is the set of all existing objects. An object O which is an instance of a class C is said to be a *member* of (or *in the extension* of) C and of all direct or indirect superclasses of C.

$$\begin{array}{|l}
(_\mathtt{isMemberOf}_) : Object \leftrightarrow Class \\
\hline
(_\mathtt{isMemberOf}_) = (_\mathtt{isInstanceOf}_) \,;\ (_\mathtt{isSubClassOf}_)^{*}
\end{array}$$

Thus, an object belongs to exactly one class as an instance and to at least one class as a member. The formal model explained up to here (depicted by the metaschema in figure 4 suffices to discuss the main problems of integrating state transition diagrams with class diagrams.

3.2 Behaviour

The *behaviour* of an object O is described by the sequence of states that it adopts during its existence. Thus, the “life” of an object is described from the object's

[2]We do not discuss object creation, object deletion, or object migration in this paper.

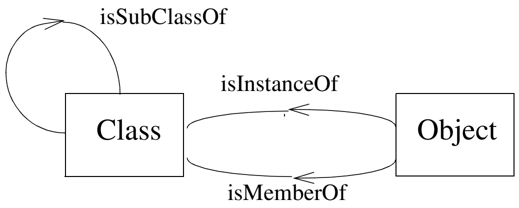


Figure 4: Metaschema

point of view. The sequence of states might be caused by method calls from several other objects (clients). The state transition diagram describes the overall behaviour of the object. All method calls are assumed to be atomic in the sense, that only the call of method is watched, not its semantics.

The behaviour of an object has (of course) to be described with respect to a class C of which O is a member. Thus, if we have O `isMemberOf` C, O is in state q, and $m \in C.M$ is a method identifier with $((q,m),\widehat{q}) \in C.STD.\delta$, then a transition of O into the state $\widehat{q}$ is possible. The resulting set of possible steps from one state to another is described by the following relation:

$$
\begin{array}{|l}
(\_ \vdash\_ \_) : Class \rightarrow (STATE \leftrightarrow STATE) \\
\hline
\forall C : Class;\ q, \widehat{q} : STATE \bullet \\
\qquad q \vdash_C \widehat{q} \\
\qquad \Leftrightarrow \\
\qquad\qquad \{q, \widehat{q}\} \subseteq C.STD.Q \wedge \\
\qquad\qquad (\exists\, m : C.STD.\Sigma \cup \{\epsilon\} \bullet \\
\qquad\qquad\qquad ((q,m),\widehat{q}) \in C.STD.\delta \vee q = \widehat{q})
\end{array}
$$

This relation describes the fact that an object belonging to a class C may move from state q to state $\widehat{q}$ if the state transition diagram of C has a corresponding arc. In addition the $\vdash_C$-relation is reflexive, i.e. an object may stay in its state in a step.

Given a class C, the relation $(\_ \vdash_C \_)$ is used to define the set $QS(C)$ of all non-empty sequences of states, which start in q_0, the start state of $C.STD$, and where every two consecutive states are connected by an arc:

$$
\begin{array}{|l}
QS : Class \rightarrow \mathbb{P}\,\mathrm{seq}_1\, STATE \\
\hline
\forall C : Class;\ qs : \mathrm{seq}_1\, STATE \bullet \\
\quad qs \in QS(C) \\
\quad \Leftrightarrow \\
\qquad qs(1) = C.STD.q_0 \wedge \\
\qquad \forall i : 1\,..\,(\#r - 1) \bullet qs(i) \vdash_C qs(i+1)
\end{array}
$$

The sequences in $QS(C)$ are the *legal sequences of states* of $C.STD$.

For instance, the set $QS(Process)$ contains as legal sequences of states sequences like

Ready Active Ready Active Terminated or

Ready Active Blocked Ready Active Terminated

or - in general - all sequences described by the regular expression

$Ready\ Active\ ([Blocked]\ Ready\ Active)^*\ Terminate.$

This definition helps to define the *legal behaviour*, i.e. the set of all possible method identifier sequences that correspond to legal state sequences.

$$
\begin{array}{|l}
(\_\mathtt{correspondsTo}\_) : Class \rightarrow (\mathrm{seq}\, MID \leftrightarrow \mathrm{seq}_1\, STATE) \\
\hline
\forall C : Class;\ ms : \mathrm{seq}\, MID;\ qs : \mathrm{seq}_1\, STATE \bullet \\
\quad ms\ \mathtt{correspondsTo}_C\ qs \\
\quad \Leftrightarrow \\
\hfill \text{[Case 1: } \#qs = 1] \\
\qquad \#qs = 1 \wedge ms = \langle\,\rangle \vee \\
\hfill \text{[Case 2: } \#qs > 1] \\
\hfill \text{[Case 2a]} \\
\qquad ((qs(1), ms(1)), qs(2)) \in C.STD.\delta \wedge \\
\qquad\quad ms(2\,..)\ \mathtt{correspondsTo}_C\ qs(2\,..) \vee \\
\hfill \text{[Case 2b]} \\
\qquad (((qs(1), \epsilon), qs(2)) \in C.STD.\delta \vee qs(1) = qs(2)) \wedge \\
\qquad\quad ms\ \mathtt{correspondsTo}_C\ qs(2\,..)
\end{array}
$$

Thus, given a state sequence qs a corresponding method sequence ms contains for each pair of consecutive states $qs(i)$ and $qs(i+1)$ a method identifier of an arc connecting those states [2a], unless there is an ϵ-arc or $qs(i) = qs(i+1)$ [2b]. In other words, ms is the sequence of method identifiers, that is collected when a path described by qs is followed through the state transition diagram.

This gives rise to the definition of the set $MS(C)$ of all legal sequences of method identifiers, which is the language accepted by the automaton $C.STD$, if all states are assumed to be acceptance states.

$$
\begin{array}{|l}
MS : Class \rightarrow \mathbb{P}\,\mathrm{seq}\, MID \\
\hline
\forall\, C : Class;\ ms : \mathrm{seq}\, MID \bullet \\
\qquad ms \in MS(C) \\
\qquad \Leftrightarrow \\
\qquad\qquad \exists\, qs : QS(C) \bullet ms\ \mathtt{correspondsTo}_C\ qs
\end{array}
$$

The states that an object O may be in and the steps given by $(\_ \vdash\_\ \_)$ that an O might perform must of course fulfill some compatibility restrictions with respect to *all* classes C with O `isMemberOf` C. Otherwise, it would not be justified to term this relation membership. The kind of restrictions, that one may pose on them, depends on the understanding of the meaning of the state transition diagram, as described in section 2.

For a given class C the set $MS(C)$ contains all possible behaviours of objects of C by describing the possible sequences of method calls. Thus, if one is interested in the effect of specialization on the behaviour of objects with respect to two classes C and C', it is the interrelation between the sets $MS(C)$ and $MS(C')$ that must be appropriately defined.

This will be discussed in more detail in section 4 on the basis of homormorphisms which will be introduced in the following section.

3.3 Homomorphisms

Let O simultaneously be a member of two classes C and C'. For instance, C could be a subclass of C' or vice versa. Then, each state with respect to C must be mapped onto a state with respect to C' in such a way that every behaviour with respect to C is reflected by a legal behaviour with respect to C'.

To achieve this we require a *homomorphism* h from the state transition diagram of C to that of C'. If STD and STD' are state transition diagrams, a function $h : STD.Q \rightarrow STD'.Q$ is a homomorphism if the following condition is fulfilled:

$$
\begin{array}{l}
\mathit{isHomomorphism} : (\mathit{STATE} \rightarrowtail \mathit{STATE}) \leftrightarrow \mathit{STDiagram} \leftrightarrow \mathit{STDiagram} \\
\hline
\forall\, h : \mathit{STATE} \rightarrowtail \mathit{STATE};\ \mathit{STD}, \mathit{STD}' : \mathit{STDiagram} \bullet \\
\quad \mathit{isHomomorphism}(h, \mathit{STD}, \mathit{STD}') \\
\quad \Leftrightarrow \\
\qquad h \in \mathit{STD}.Q \rightarrow \mathit{STD}'.Q \wedge \\
\qquad \forall\, q, \widehat{q} : \mathit{STD}.Q;\ m : \mathit{STD}.\Sigma \cup \{\epsilon\} \bullet \\
\qquad\quad ((q, m), \widehat{q}) \in \mathit{STD}.\delta \\
\qquad\quad \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\ m \text{ is also known in } \mathit{STD}'] \\
\qquad\qquad m \in \mathit{STD}'.\Sigma \wedge \\
\qquad\qquad\quad ((h(q), m), h(\widehat{q})) \in \mathit{STD}'.\delta \vee \qquad\qquad \text{[a]} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\ m \text{ is not known in } \mathit{STD}'] \\
\qquad\qquad m \notin \mathit{STD}'.\Sigma \wedge \\
\qquad\qquad\quad (((h(q), \epsilon), h(\widehat{q})) \in \mathit{STD}'.\delta \vee \qquad\qquad \text{[b1]} \\
\qquad\qquad\quad h(q) = h(\widehat{q})) \wedge \qquad\qquad\qquad\qquad \text{[b2]} \\
\qquad h(\mathit{STD}.q_0) = \mathit{STD}'.q_0
\end{array}
$$

The three different cases [a], [b1], and [b2] are illustrated in figure 5. A homomorphism maps the states of the corresponding diagrams in such a way that all arcs are kept, which are annotated by method identifiers known in *C and C′*. Arcs which are annotated by identifiers only known in C are either mapped to ϵ-arcs in $C'.STD$ or have their incident vertices mapped onto the same vertex in $C'.STD$.

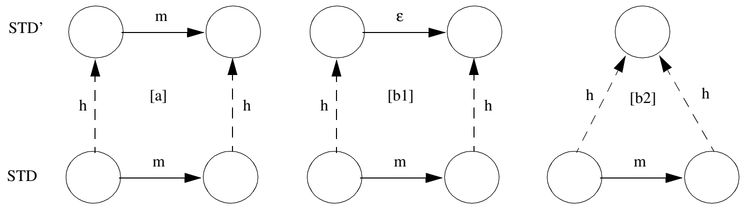


Figure 5: STDiagram Homomorphisms

This very abstract notion of homomorphism covers a variety of possible practical cases as will be exemplified in section 4.

Using this definition, we can show that each legal behaviour with respect to a class C can be mapped onto a legal behaviour with respect to a class C' if a homomorphism between their respective state transition diagrams exists.

Theorem 1

$$
\begin{array}{l}
\forall\, C, C' : Class;\ h : STATE \nrightarrow STATE \bullet \\
\quad isHomomorphism(h, C.STD, C'.STD) \\
\quad \Rightarrow \\
\qquad \forall\, q, \widehat{q} : C.STD.Q \bullet \\
\qquad\quad q \vdash_C \widehat{q} \Rightarrow h(q) \vdash_{C'} h(\widehat{q}).
\end{array}
$$

Proof

Assume

$$q \vdash_C \widehat{q},$$

which means

$$
\begin{array}{l}
\{q, \widehat{q}\} \subseteq C.STD.Q \wedge \\
(\exists\, m : C.STD.\Sigma \cup \{\epsilon\} \bullet ((q, m), \widehat{q}) \in C.STD.\delta \vee q = \widehat{q}).
\end{array}
$$

Since h is a homomorphism, this implies

$$
\begin{array}{l}
\{h(q), h(\widehat{q})\} \subseteq C'.STD.Q \wedge \\
(\exists\, m : C.STD.\Sigma \cup \{\epsilon\} \bullet \\
\quad m \in C'.STD.\Sigma \wedge \\
\qquad ((h(q), m), h(\widehat{q})) \in C'.STD.\delta \vee \\
\quad m \notin C'.STD.\Sigma \wedge \\
\qquad (((h(q), \epsilon), h(\widehat{q})) \in C'.STD.\delta \vee \\
\qquad h(q) = h(\widehat{q})) \vee \\
h(q) = h(\widehat{q}))
\end{array}
$$

which easily simplifies to the definition of

$$h(q) \vdash_{C'} h(\widehat{q}).$$

$\square$

This result holds equally for the set of legal state sequences. Thus, we have:

Corollary 1

$$qs \in QS(C) \Rightarrow h \circ qs \in QS(C').$$

$\square$

Corollary 1 states that all state sequences with respect to a class C are reflected by state sequences with respect to any other class C', as long as there exists a homomorphism from the state transition diagram of class C to that of class C'. In section 4 we will present several examples for such homomorphisms.

As described above, our main goal is to describe the interrelation of the method sequences $MS(C)$ and $MS(C')$. This interrelation can be derived from the interrelation of state sequences.

Theorem 2

$$\begin{array}{l} \forall\, C, C' : Class;\ h : STATE \nrightarrow STATE \bullet \\ \quad isHomomorphism(h, C.STD, C'.STD) \\ \qquad \Rightarrow \\ \qquad\quad (ms \in MS(C) \Rightarrow ms \upharpoonright C'.M \in MS(C')). \end{array}$$

Proof

Assume there are sequences $ms \in MS(C)$ and $qs \in QS(C)$
s.t. ms $\texttt{correspondsTo}_C$ qs.

Following corollary 1 there is also a sequence $qs' \in QS(C')$ with $qs' = h \circ qs$, i.e. $\forall\, i : 1 \mathinner{..} \#qs \bullet qs'(i) = h(qs(i))$.

Let

$$ms' == ms \upharpoonright C'.M.$$

According to the definition of the ($\_\texttt{correspondsTo}\_$)-relation there are three cases to be distinguished.

Case 1:
If $\#qs' = 1$, we have also $\#qs = 1$ and thus $ms = \langle\,\rangle = ms'$.

Case 2:
If $\#qs' > 1$, we have a look at the first transition in qs:

> **Case 2a**:
> If $((qs(1), ms(1)), qs(2)) \in C.STD.\delta$
> then we have either
> $ms(1) \in C'.M$
> and thus
> $ms(1) = ms'(1)$
> or
> $ms(1) \notin C'.M$
> in which case the definition of homomorphism implies:
> $((qs'(1), \epsilon), qs'(2)) \in C'.STD.\delta \vee qs'(1) = qs'(2)$.
>
> **Case 2b**:
> If $((qs(1), \epsilon), qs(2)) \in C.STD.\delta \vee qs(1) = qs(2)$
> we also have $((qs'(1), \epsilon), qs'(2)) \in C'.STD.\delta \vee qs'(1) = qs'(2)$.

Altogether this leads to

$$ms' = \langle\,\rangle \wedge \#qs' = 1 \vee \qquad [1]$$
$$ms(1) \in C'.M \wedge$$
$$((qs'(1), ms'(1)), qs'(2)) \in C'.STD.\delta \vee \qquad [2a]$$
$$ms(1) \notin C'.M \wedge$$
$$(((qs'(1), \epsilon), qs'(2)) \in C'.STD.\delta \vee qs'(1) = qs'(2)). \qquad [2b]$$

Using
$ms'(2..)$ `correspondsTo`$_{C'}$ $qs'(2..)$
in case 2a and
ms' `correspondsTo`$_{C'}$ $qs'(2..)$
in case 2b as the hypotheses this leads via induction to

ms' `correspondsTo`$_{C'}$ qs'.

□

Theorem 2 states that the existence of a homomorphism between two classes C and C' is a sufficient condition for the compatibility of the corresponding life cycles. Section 4 will apply this result to the context of behaviour descriptions.

4 Two Kinds of Behaviour

In this section we explain that the existence of certain homomorphisms implies compatibility of the integration of the object model and the dynamic model with specialization. The notion of homomorphism and Theorem 2 will be used to formulate sufficient conditions, which imply observability or invocability, respectively.

We shall also give examples of transformation rules for constructing state transition diagrams for subclasses from those of a superclass. The correctness of these rules is proved by showing that an appropriate homomorphism exists for each of them. Thus, we give constructive ways to build state transition diagrams which fulfill the declarative conditions from section 3. These might be used as guidelines for deriving subclass state transition diagrams from given state transition diagrams in practice.

4.1 General Assumption

As described above, there are two ways of thinking about state transition diagrams as a means of describing the behaviour of objects of a certain class C. Thus, when describing a class C one can choose between two different approaches leading to *ObsC* or *InvC*. Here *ObsC* describes the class using the observability approach and *InvC* describes it according to the invocability approach. Usually the state transition diagrams of *ObsC* and *InvC* will differ.

To show the difference between *ObsC* and *InvC* consider as an example the state transition diagram for a TV set in figure 3, where its behaviour was described from the invocability point of view. Figure 6 describes the TV set from the observability point of view. Here additional ϵ-arcs allow "‘spontaneous"’ transitions, which are specialized in the class *TvRemote*.

Let the method identifier sequences described by those two class behaviour de-

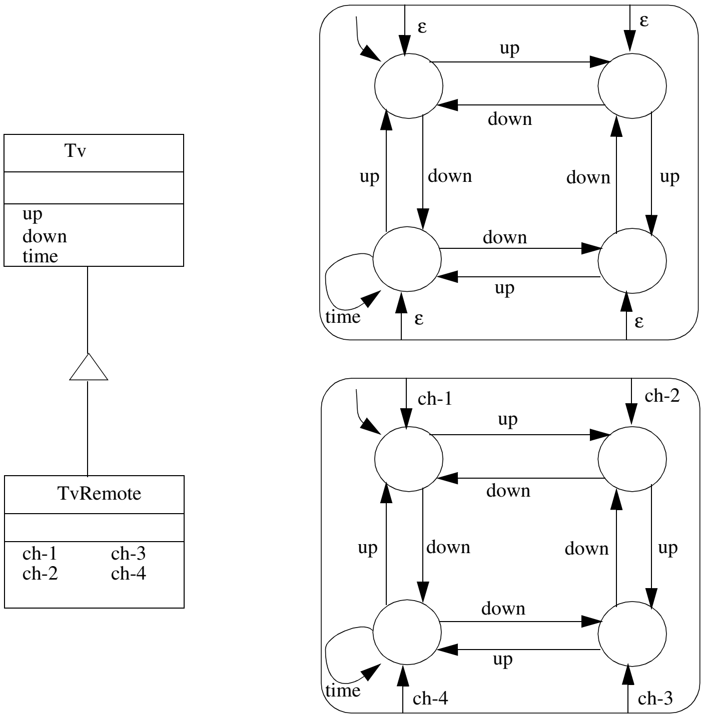


Figure 6: Observable Behaviour of the TV Set

scriptions be $OS(C) = MS(ObsC)$ and $IS(C) = MS(InvC)$, respectively. Apparently, as a consistency condition, we have the general assumption that $IS(C) \subseteq OS(C)$ should hold.

Furthermore, it is natural that the state transition diagram *ISTD* describing the invocability is a subdiagram of the state transition diagram *OSTD* describing the observability. Since inclusion of the state transition diagrams implies inclusion of the sequences, the general assumption from above is fulfilled in that case.

4.2 Observability

In section 2 we introduced the observability approach. To obtain consistency of observability state transition diagrams with specialization, the aim is to find conditions that guarantee for a subclass C of a superclass C' that all observable sequences in C are also observable in C', as far as the methods of C' are concerned, short:

$$OS(C) \upharpoonright C'.M \subseteq OS(C')$$

Theorem 2 in section 3.3 states that the existence of a homomorphism from C to C' implies this consistency, more formally:

▷ if state transition diagrams are viewed as a means to describe the observable behaviour, the following condition should be fulfilled:

$$\begin{array}{l} \forall\, C, C' : Class \mid C \ \texttt{isSubClassOf}\ C' \bullet \\ \quad \exists\, h : C.STD.Q \to C'.STD.Q \bullet \\ \quad\quad isHomomorphism(h, C.STD, C'.STD) \end{array}$$

There are several ways to achieve the existence of such homomorphisms. This can be done by defining transformation rules for deriving a specialized subclass state transition diagram from a given diagram in such a way that the existence of an according homomorphism is guaranteed.

The following figures contain some examples of possible transformation rules on superclass state transition diagrams to build subclass state transition diagrams which are compatible from the observability point of view.

1. It is allowed to refine a state of the superclass by a whole subdiagram in the subclass, as long as there are only new method identifiers used inside the

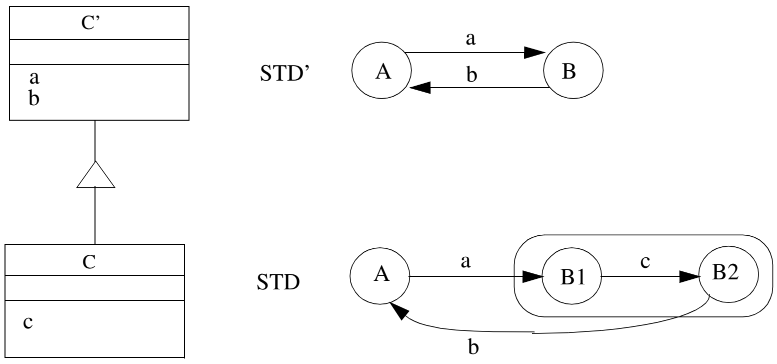
subdiagram (cf. figure 7).
By mapping all states of the subdiagram onto the refined state, a homomorphism is yielded.

2. The superclass diagram may be parallely extended by another diagram leading to an AND-superstate in the sense of statecharts (cf. figure 8).
   Here the projection of the subclass diagram to the original diagram is a homomorphism.

3. Transitions and/or states of the superclass diagram may be deleted (cf. figure 9).
   In this case the embedding function is an appropriate homomorphism.

4. It is possible to replace ϵ-transitions in the superclass diagram by transitions labeled with new methods (cf. figure 10).
   Then, the identity function has the homomorphism property.

Many more cases, especially combinations of these cases are possible.



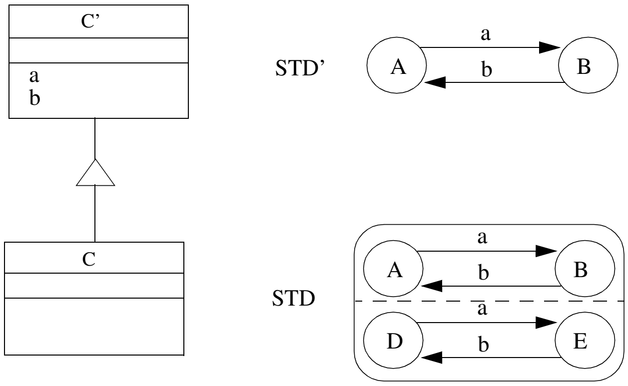
Homomorphism

h : STD.Q $\rightarrow$ STD'.Q

h	A	B1	B2
	A	B	B

Figure 7: Refinement of States

The observability approach goes hand in hand with the use of ϵ-transitions. If there are transitions which may be observed with instances of a subclass but which are

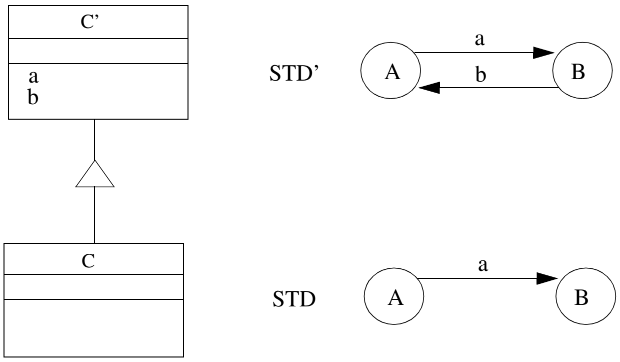


Homomorphism

h : STD.Q $\rightarrow$ STD’.Q

h	(A,D)	(A,E)	(B,D)	(B,E)
	A	A	B	B

Figure 8: Parallel Extension



Homomorphism

h : STD.Q $\rightarrow$ STD’.Q

h	A	B
	A	B

Figure 9: Deletion of States and/or Transitions

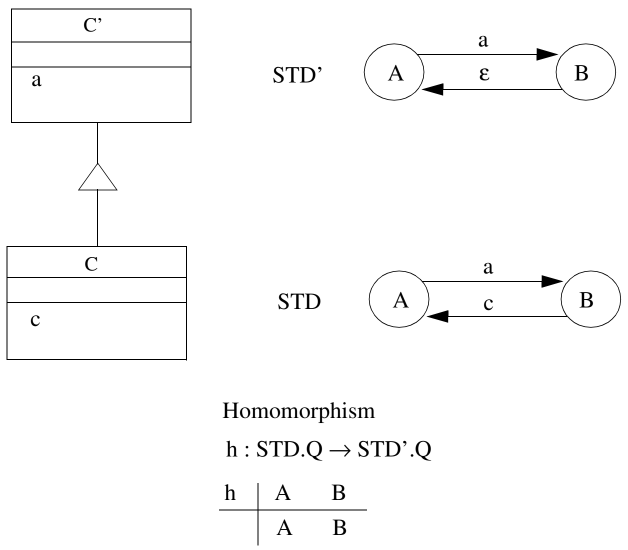


Figure 10: Replacement of ϵ-Transitions

not invocable by a method, an ϵ-transition is an appropriate means to describe the effect that a state changes spontaneously. Views of objects built by restricting the interface to a proper subset of their methods is a special case ([EbeEng 94]). Also, ϵ-transitions might be used as a place holder (in the sense of a virtual method) for a later refinement of the behaviour in the specialized classes.

4.3 Invocability

Similarly, sufficient conditions for consistency of invocability state transition diagrams with specialization can be given. Here, the aim is to find conditions that guarantee, that all invocable sequences in a superclass C' are also invocable in any subclass C, short:

$$IS(C') \subseteq IS(C)$$

Thus, we have

▷ if state transition diagrams are used as prescriptions of the invocable behaviour, a sufficient condition is:

$$\begin{array}{l}\forall\, C, C' : \mathit{Class} \mid C \; \texttt{isSubClassOf} \; C' \bullet \\ \quad \exists\, h : C'.\mathit{STD}.Q \rightarrow C.\mathit{STD}.Q \bullet \\ \qquad \mathit{isHomomorphism}(h, C'.\mathit{STD}, C.\mathit{STD})\end{array}$$

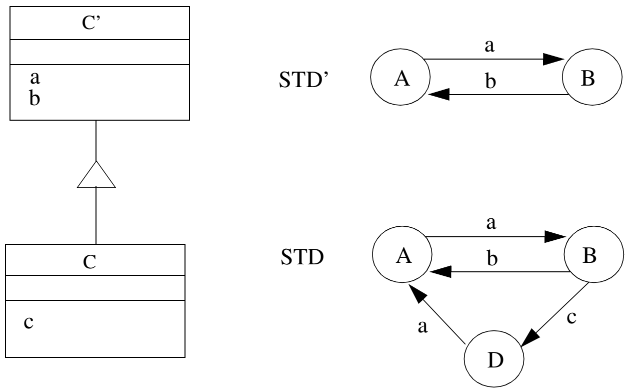
Then, theorem 2 implies that the existence of a homomorphism from the superclass state transition diagram to the subclass state transition diagram implies the desired inclusion.

The following figures contain some example for transformation rules which imply homomorphisms of this kind.

1. The superclass diagram may be extended in any form as long as the new diagram contains the original state transition diagram as an embedded subdiagram (cf. figure 11). The embedding function is the appropriate homomorphism.
2. As a special case the extension by a parallel diagram with disjoint method identifiers preserves invocability, too (cf. figure 12).
3. It is possible to fuse subdiagrams into a single state as long as all transitions are replaced by appropriate loops (cf. figure 13).
   The function that maps all fused states into the target state is the homomorphism.

4.4 Summary

If the approach discussed above is used in its full extension, then each class description in the object model can be accompanied by two different state transition diagrams, which lead to different behaviours $OS(C)$ and $IS(C)$. Then, to achieve consistency with specialization there should exist homomorphisms from the observability state transition diagrams of the subclasses to those of their respective superclasses, and on the other hand there should exist homomorphisms from the invocability state transition diagrams of superclasses to those of their subclasses.

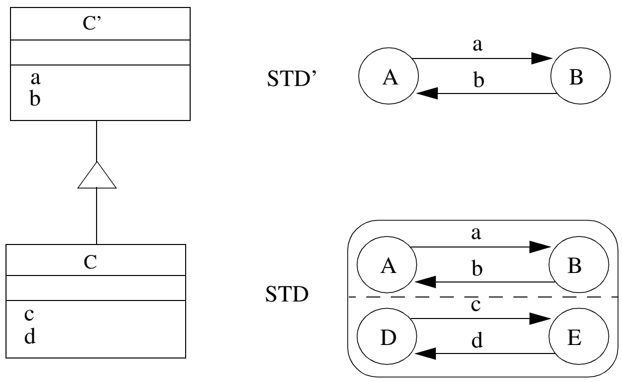


Homomorphism

$h : STD'.Q \rightarrow STD.Q$

h	A	B
	A	B

Figure 11: Extensions by States and/or Transitions

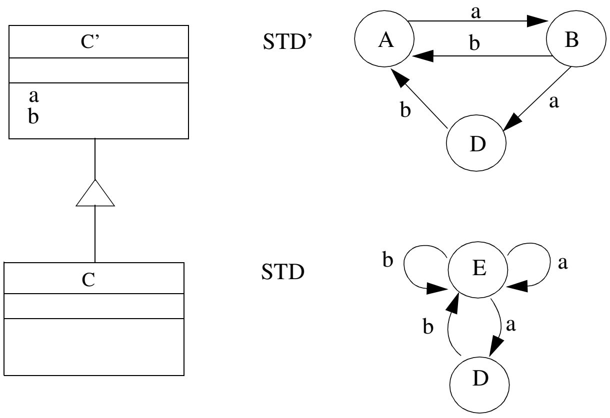


Homomorphism

$h : STD'.Q \rightarrow STD.Q$

h	A	B
	(A,D)	(B,D)

Figure 12: Parallel Extension



Homomorphism

h : STD'.Q → STD.Q

h	A	B	D
	E	E	D

Figure 13: Fusion of Subdiagrams

Looking at the different class descriptions we then have the following inclusions for the sets of legal method sequences of a subclass C with respect to a superclass C':

$IS(C') \subseteq IS(C) \subseteq OS(C)$
and
$(OS(C) \restriction C'.M) \subseteq OS(C')$.

5 Related Work

The problem of specializing state transition diagrams is only informally described (if at all) in most of the books on object-oriented analysis and design methodes, e.g. [RuBlPr 91] or [ShlMel 92].

[SaHaJu 94] use state charts as life cycle diagrams in a notation according to the spirit of OMT. Their approach coincides with our observability approach. They assume the existence of morphisms between the state transition diagrams and characterize them operationally as sequences of primitive manipulations to the superclass state transition diagram, all of which are covered by the homomorphism approach in this paper. [SaHaJu 94] includes also preconditions into their state charts, which we did not in order to focus on our main results.

[LopCos 93] also approach the problem from the observability point of view. They claim that the subclass state transition diagram should be a refinement. They define refinement by the existence of so-called *inheritance morphisms* from subclasses to superclasses. Their morphisms differ from our homomorphisms by excluding case (b) from our definition. Thus, they are more restrictive (e.g. our definition includes also views as special cases, see [EbeEng 94]). In [LopCos 93] the authors prove in a graph grammar approach that their inheritance morphisms are equivalent to applying node expansion, arc elimination and/or isolated node elimination to the superclass state transition diagram.

McGregor and Dyer ([McGDye 93]) discuss also techniques for the construction of subclass state transition diagrams from those of a superclass, apparently following an observability approach. They use state refinement and parallel extension as the main operations. But their approach is only partially formalized.

A detailed discussion of so-called conformance of behaviour between superclasses and subclasses is given in [CooDan 94] with the Syntropy method. Here the authors require that "‘every event sequence accepted by an object of a supertype, leaving it in a particular state, will be accepted by an object of the subtype and leave in the same state.’' This definition apparently describes an invocability approach to behaviour specialization. Their approach is based on general statecharts and includes also several kinds of annotations like precondition, postconditions, generated events etc. They verbally give 13 rules which explain how statecharts of supertypes can be extended to yield conformant statecharts of subtypes. All rules concerning flat state transition diagrams can be stated formally and unambigously using our approach by defining the appropriate set of homomorphisms. Thus, they turn out to be special cases of invocability descriptions. The one rule they give (rule 6) using intrinsic properties of statecharts can be covered too, if the notion of homomorphism would be extended to statecharts.

[Rump 96] und [PaeRum 94] define refinement of state transition diagrams on a semantical level and [Rump 96] gives several rules which imply semantical refinement. Their approach is also a special case of observability.

The problem of how to extend the life cycle description of a superclass in a subclass has also been intensively studied in the field of concurrent object-oriented programming languages ([MatYon 93]). Class definitions in a concurrent object-oriented program contain definitions how incoming method invocations have to be synchronized, which corresponds to observability. Since synchronization constraints defined in a superclass have to be refined within a subclass to handle also newly defined methods, they have to be explicitly known and modified within the subclass definition. This contradicts the intuitive idea of encapsulation and is here called "‘inheritance anomaly"’ ([MatYon 93]). In our approach, we consider life cycle descriptions of classes as public, as it is commonly done in the world of object-oriented modeling.

Besides state transition diagrams other approaches for describing the behaviour may be used, e.g. Petri nets ([KapSch 94]). [SchStu 95] describe our distinction of observable and invocable behaviour in this context. Other approaches are path expressions ([CamHab 74]), process algebra ([BaeWei 90]), temporal logic ([JuSaHa 91]). Since our approach inherently uses non-deterministic state diagrams, it is not immediately transferable to each of these paradigms.

6 Conclusion

By introducing a formal model and a general form of automata homomorphisms for life cycle diagrams, we showed that integration of structural and dynamic descriptions can be made consistent with inheritance.

The homomorphism conditions depend on the approach used for defining the behaviour, which is different if observability or invocability are to be described. For both approaches we gave several examples.

It should be the task of the method engineer to incorporate one of these approaches into a software development method to be defined depending on his or her view. In extreme cases this might even result in a method where both approaches are used simultaneously. This means that two different state transition diagrams have to be defined for every class.

The formalization and the transformations discussed in the paper also help to build software development tools to support the incremental development of object-oriented analysis and/or design descriptions which consistently integrate structural and dynamic aspects. In particular the tool can be made configurable with respect to the approach chosen. Depending on the approach modification of subclass diagrams can be restricted accordingly. By keeping the homomorphism in an internal data structure the specialization can be explicitly traced.

In this paper we presented our results on the basis of flat state transition diagrams. A more elaborated framework can be developed analogously for statecharts in a straight-forward manner by extending the notion of homomorphism appropriately.

Acknowledgements We thank the anonymous referees and the following colleagues for some helpful discussions during the course of this work: Michael von der Beeck, Gerti Kappel, Gunter Saake, Michael Schrefl, Roel Wieringa.

References

[BaeWei 90] J.C.M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge University Press, Cambridge, 1990.

[BoRuJa 98] Grady Booch, James Rumbaugh, Ivar Jacobson, *Unified Modeling Language User Guide*, Addison Wesley, Reading MA, 1998 (to appear).

[CamHab 74] R.H. Campbell, A.N. Habermann, *The Specification of Process Synchronization by Path Expressions*, Springer, Berlin, 1974, LNCS 16, pp. 89-102.

[CooDan 94] Steve Cook, John Daniels, *Designing Object Systems - Object-Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994.

[EbeEng 93] Jürgen Ebert, Gregor Engels, *Design Representation*, in: J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, Wiley, New York, 1993, pp.382-394.

[EbeEng 94] Jürgen Ebert, Gregor Engels, *Structural and Behavioural Views on OMT-Classes*, in: Elisa Bertino, Susan Urban (Eds.), *Object Oriented Methodologies and Systems*, Springer, Berlin, 1994, LNCS 858, pp.142-157.

[Ebert 93] *Efficient Interpretation of State charts.* in: Zolt'an 'Esik (Ed.), *Fundamentals of Computation Theory (FCT'93)* Springer, Berlin, 1993, LNCS 710, pp.212-221.

[EmKuWo 92] David W. Embley, Barry D. Kurtz, Scott N. Woodfield, *Object-Oriented System Analysis - A Model-Driven Approach*, Yourdon Press, Prentice Hall, Englewood Cliffs, 1992.

[Har 87] David Harel, *Statecharts: a Visual Formalism for Complex Systems*, Science of Computer Programming 8 (1987,3), 231-274.

[HopUll 79] John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading MA, 1979.

[Jaco 92] Ivar Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, Wokingham, 1992.

[JuSaHa 91] Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, Cristina Sernadas, *Object-Oriented Specification of Information Systems: The TROLL Language*, Technische Universiät Braunschweig, Technical Report 91-04.

[KapSch 94] Gerti Kappel, Michael Schrefl, *Inheritance of Object Behavior - Consistent Extensions of Object Life Cycles*, in: J. Eder, L. Kalinichenko (eds.), *Extending Information Systems Technology, Proceedings of the Second International East/West Database Workshop*, Springer, Berlin, WSCS, 1994.

[KhoAbn 90] Setrag Khoshafian, Razmik Abnous, *Object Orientation - Concepts, Languages, Databases, User Interfaces*, John Wiley, New York, 1990.

[LopCos 93] A.Lopes, J.F.Costa, *Rewriting for Reuse*, in: *Proceedings ERCIM Workshop on Development and Transformation of Programs*, INRIA, Nancy, Nov. 1993, pp.43-55.

[MatYon 93] Satoshi Matsuoka, Akinori Yonezawa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in: G. Aga, P. Wegner, A. Yonezawa (Eds.), *Research Directions in Concurrent Object-Oriented Programming*, ACM Press, New York, 1993.

[Meye 88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, 1988.

[McGDye 93] J.D.McGregor, D.M. Dyer, *A Note on Inheritance and State Machines*, *ACM Software Engineering Notes*, Vol. 18, No. 4, Oct. 1993, pp. 61-69.

[PaeRum 94] Barbara Paech, Bernhard Rumpe, *A new Concept of Refinement used for Behaviour Modelling with Automata*, in *Formal Methods Europe (FME'94)*, Springer, Berlin, 1994, LNCS 873.

[RuBlPr 91] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood

Cliffs NJ, 1991.

[Rump 96] Bernhard Rumpe, *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*, Heribert Utz Verlag, München, 1996

[SaHaJu 94] Gunter Saake, Peter Hartel, Ralf Jungclaus, Roel Wieringa, Remco Feenstra, *Inheritance Conditions for Object Life Cycle*, in: Udo W. Lipeck, Gottfried Vossen (Hrsg.), *Formale Grundlagen für den Entwurf von Informationssystemen*, Universität Hannover, Informatik-Bericht 03/94, pp.79-88.

[SchStu 95] Michael Schrefl, Markus Stumptner, *Behavior Consistent Extension of Object Life Cycles*, in: Michael Papazoglou (Ed.), *OOER'95: Object-Oriented and Entity-Relationship Modeling*. Springer, Berlin, 1995, LNCS 1021.

[ShlMel 92] S. Shlaer, St.J. Mellor, *Object Lifecycles: Modeling the world in state*, Yourdon Press, Englewood Cliffs NJ, 1992.

[Spiv 92] J.M. Spivey, *The Z Notation (2nd Edition)*, Prentice Hall, New York, 1992.