Specialization of Object Life Cycle Definitions

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

Gregor Engels  
Leiden University  
Dept. of Computer Science  
P.O. Box 9512  
NL-2300 RA Leiden  
phone: ++31-71-527-7069  
fax: ++31-71-527-6985  
engels@wi.leidenuniv.nl

January 1997

Abstract

Several object-oriented modeling approaches propose to describe the dynamic behaviour of objects by state transition diagrams. None of them provides precise rules or conditions for the interrelation between the behaviour description of classes and those of their subclasses.

In this paper, we discuss this interrelation in detail. It turns out that one has to distinguish between the observable and the invocable behaviour of objects and that different compatibility requirements between the diagrams exist depending on the type of behaviour.

Keywords: object model, dynamic model, object life cycle, state transition diagram, 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 ([RuBIPr 91]) or OOA ([ShlMel 92]).

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 transi-
tion 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. This paper is an extension and generalization of the results presented previously [EbeEng 94]. While that paper was re-
stricted to the first kind of modeling behaviour, we here discuss both aspects of dynamic behaviour inheritance 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 $C$:\footnote{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]).}

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

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 dynamic 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. [?]).)

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*. \hfill \Box

**Example 2.** As a second example, we give a subclass definition *LoggingProcess* in figure 2, where the corresponding state transition diagram is constructed
Figure 1: STDiagram of a subclass constructed by parallel extension
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 get dequeued. Therefore, there exist invocable life cycles of a Process that are not possible for a LoggingProcess. This leads to the second kind of behaviour inheritance, 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 an inverse relation with respect to inheritance:

> each sequence which is invocable (or executable) with respect to a given class 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 \( Tvl \) (television) and of a corresponding subclass \( TvlRemote \) (television with remote control). Figure 3 gives the class definition as well as the dynamic behaviour description of these two classes. The class \( Tvl \) 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 \( TvlRemote \) has in addition a remote control with four buttons \( ch-i \), which enable to choose 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*

\[ \text{up } \text{ch-4} \text{ time up} \]

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

\[ \text{up time up} \]

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

### 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 into
Figure 3: STDiagram of a subclass constructed by adding transitions
the description.

At first, we shall define the object model including inheritance 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 dynamic 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 inheritance (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])\(^2\) assuming some universes of identifiers, object identifiers and states.

\[ ID, OID, STATE \]

Here, \( ID \) stands for all possible method identifiers, \( OID \) is the stock of all possible objects described by their respective 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 \( \Sigma \) of identifiers, and a relation \( \delta \) which describes possible state changes together with their possible

\(^2\)We assume that readers are familiar with the basisc of Z are assumed. If the journal editors want it, an appendix about Zcould be added, cf. first article in ACM TOSEM 4(1995,4).
inputs. \( Q \) stands for the vertices in the diagram and \( \delta \) denotes its arcs. Here, the transition relation \( \delta \) is non-deterministic and allows spontaneous \( \epsilon \)-transitions.

\[
\begin{array}{ll}
\text{STDDiagram} \\
Q : \mathbb{F} \text{STATE} & \text{[states]} \\
\Sigma : \mathbb{F} \text{ID} & \text{[input alphabet]} \\
\delta : \mathbb{F} \left( (Q \times (\Sigma \cup \{\epsilon\})) \times Q \right) & \text{[transition relation]} \\
q_0 : Q & \text{[start state]}
\end{array}
\]

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 \( \epsilon \)-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 describe the integration of structure and dynamics, 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.

\[
\begin{array}{ll}
\text{Class} \\
M : \mathbb{F} \text{ID} & \text{[method identifiers]} \\
\text{STD} : \text{STDDiagram} & \text{[behaviour]} \\
\text{STD.} \Sigma = M
\end{array}
\]

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

\[
\begin{array}{ll}
(\text{isSubClassOf}_\_): \text{Class} \leftrightarrow \text{Class} \\
\text{isAcyclic}(\text{isSubClassOf}_\_) \\
\forall C, C' : \text{Class} \mid C \text{ isSubClassOf } C' \Rightarrow C' \cdot M \subseteq C \cdot M
\end{array}
\]
There may exist a set of corresponding objects for each class description \( C \) at a given point of time\(^3\). Objects are represented by their respective object identifiers which stem from the common universe \( OID \). Each existing object \( oid \) belongs to exactly one class. It is said to be an instance of that class, i.e. there is a partial function \( \text{asInstanceOf} \) assigning class descriptions to object identifiers.

\[
\text{asInstanceOf} : OID \rightarrow \text{Class}
\]

The domain of \( \text{asInstanceOf} \) is the set of all existing objects. An object \( oid \) 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 \( C \).

\[
\begin{align*}
\text{memberOf} & : OID \leftrightarrow \text{Class} \\
\text{memberOf} = \text{asInstanceOf}; \text{memberOf'}
\end{align*}
\]

Thus, objects belong to exactly one class as instances and to several classes as members. The formal model explained up to here suffices to discuss the main problems of integrating state transition diagrams with class diagrams.

### 3.2 Behaviour

The behaviour of an object \( oid \) is described by the sequence of situations that it adopts during its existence. In the simple model used here, situations coincide with the states of the state transition diagrams. Thus, the “life” of an object is described from the object’s point of view. The sequence of situations 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 \( oid \) is a member. Thus, if we have \( oid \text{memberOf} C \), \( oid \) is

\(^3\)We do not discuss object creation, object deletion, or object migration in this paper.
in state \( q \), and \( m \in C.M \) is a method identifier with \( ((q, m), \hat{q}) \in C.STD.\delta \) then a transition of \( oid \) into the state \( \hat{q} \) is possible. The resulting set of possible steps from one situation to another is described by the following relation:

\[
\forall C : \text{Class}; \ q, \tilde{q} : \text{STATE} \bullet \\
q \vdash_C \tilde{q} \\
\Leftrightarrow \\
\{ q, \tilde{q} \} \subseteq C.STD.Q \land \\
(\exists m : C.STD.\Sigma \cup \{ \epsilon \} \bullet \\
(q, m), \tilde{q}) \in C.STD.\delta \land \\
q = \tilde{q}
\]

This relation describes the fact that an object belonging to a class \( C \) may move from state \( q \) to state \( \tilde{q} \) 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 \( (- \vdash_c -) \) implicitly defines the set \( R(C) \) of all legal sequences of states of \( C.STD \) which may occur:

\[
R : \text{Class} \rightarrow \mathbb{P} \text{seq STATE} \\
\forall C : \text{Class}; \ r : \text{seq}_1 \text{STATE} \bullet \\
r \in R(C) \\
\Leftrightarrow \\
r(1) = C.STD.q_0 \land \\
\forall i : 1 \ldots (#r - 1) \bullet r(i) \vdash_C r(i + 1)
\]

The sequences in \( R(C) \) are 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.

For instance, the set \( R(\text{Process}) \) contains as legal sequences of states sequences like

\textit{Ready Active Ready Active Terminated} or

14
or - in general - all sequences described by the regular expression

\[ \text{Ready Active (Blocked) Ready Active}^* \text{ Terminate}. \]

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

\[
\text{\_correspondsTo\_} : \text{Class} \rightarrow (\text{seq ID} \leftrightarrow \text{seq1 STATE})
\]

\[
\forall C \in \text{Class}; \ s : \text{seq ID}; \ r : \text{seq1 STATE} \bullet
\]

\[
\text{s correspondsTo}_C \ r
\]

\[ \iff \]

\[
s = () \land \#r = 1 \lor
\]

\[
((r(1), s(1)), r(2)) \in C.\text{STD} \land
\]

\[
s(2..) \text{ correspondsTo}_C \ r(2..) \lor
\]

\[
((r(1), \epsilon), r(2)) \in C.\text{STD} \lor r(1) = r(2) \land
\]

\[
s \text{ correspondsTo}_C \ r(2..)
\]

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

This gives rise to the definition of the set \( S(C) \) of all legal sequences of method identifiers, which is the language defined by the automaton \( C.\text{STD} \) according to [HopUll 79].

\[
S : \text{Class} \rightarrow \mathcal{P} \text{seq ID}
\]

\[
\forall C \in \text{Class}; \ s : \text{seq ID} \bullet
\]

\[
s \in S(C)
\]

\[ \iff \]

\[
\exists r : R(C) \bullet s \text{ correspondsTo}_C \ r
\]

The situations that an object \( \text{oid} \) may be in and the steps (given by \( \_ \vdash \_ \_ \) ) that an \( \text{oid} \) might perform must of course fulfill some compatibility restrictions with respect to all classes that \( \text{oid} \) is a member of. The kind of re-
strictions, 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 \( S(C) \) contains all possible behaviours of objects
of \( C \) by describing the allowed sequences of method calls. Thus, if one is
interested in the effect of inheritance on the behaviour of objects with respect
to two classes \( C \) and \( C' \), it is the interrelation between the sets \( S(C) \) and
\( S(C') \) that must be appropriately defined.

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

### 3.3 Homomorphisms

Let \( oid \) simultaneously be a member of two classes \( C \) and \( C' \). For instance,
\( C \) could be a subclass of \( C' \) or vice versa. Then, each situation with respect
to \( C \) must be mapped onto a situation 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 dia-
gram 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:
isHomomorphism : (STATE → STATE) ⇒ STD\text{Diagram} ⇔ STD\text{Diagram} 

∀ h : STATE → STATE; STD, STD' : STD\text{Diagram} • 
isHomomorphism(h, STD, STD') 
⇔ 
h ∈ STD.Q → STD'.Q ∧ 
∀ q, q' : STD.Q; m : STD.Σ ∪ \{ε\} • 
((q, m), q') ∈ STD.δ 
⇒ 
m ∈ STD'.Σ ∧ 
((h(q), m), h(q')) ∈ STD'.δ ∨ [a] 
\forall m, q' : STD'.Σ • 
(((h(q), ε), h(q')) ∈ STD'.δ ∨ [b] 
h(q) = h(q') ∧ [c] 
h(STD.q₀) = STD'.q₀ 

The three different cases [a] - [c] are illustrated in figure 4. 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 4: STD\text{Diagram} Homomorphisms](image)

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

\[ \forall C, C': \text{Class}; \; h : \text{STATE} \to \text{STATE} \bullet
\]

\[ \text{isHomomorphism}(h, C.\text{STD}, C'.\text{STD}) \]

\[ \Rightarrow \]

\[ \forall q, \hat{q} : C.\text{STD}.Q \bullet
\]

\[ q \vdash_C \hat{q} \Rightarrow h(q) \vdash_{C'} h(\hat{q}). \]

Proof

Assume

\[ q \vdash_C \hat{q}. \]

which means

\[ \{ q, \hat{q} \} \subseteq C.\text{STD}.Q \land \]

\[ (\exists m : C.\text{STD}.\Sigma \cup \{ \epsilon \} \bullet ((q, m), \hat{q}) \in C.\text{STD}.\delta \lor q = \hat{q}). \]

Since \( h \) is a homomorphism, this implies

\[ \{ h(q), h(\hat{q}) \} \subseteq C'.\text{STD}.Q \land \]

\[ (\exists m : C'.\text{STD}.\Sigma \cup \{ \epsilon \} \bullet \]

\[ m \in C'.\text{STD}.\Sigma \land \]

\[ ((h(q), m), h(\hat{q})) \in C'.\text{STD}.\delta \lor \]

\[ m \not\in C'.\text{STD}.\Sigma \land \]

\[ ((h(q), \epsilon), h(\hat{q})) \in C'.\text{STD}.\delta \lor \]

\[ h(q) = h(\hat{q}) \lor \]

\[ h(q) \vdash_{C'} h(\hat{q}). \]

which easily simplifies to the definition of

\[ h(q) \vdash_{C'} h(\hat{q}). \]

This result holds equally for the set of legal state sequences. Thus, we have:
Corollary 1

\[ r \in R(C) \Rightarrow h \circ r \in R(C'). \]

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 \( S(C) \) and \( S(C') \). This interrelation can be derived from the interrelation of state sequences.

Theorem 2

\[
\forall C, C': \text{Class}; \quad h: \text{STATE} \rightarrow \text{STATE} \bullet
\]
\[\text{isHomomorphism}(h, C.\text{STD}, C'.\text{STD}) \Rightarrow \]
\[ (s \in S(C) \Rightarrow s \upharpoonright C'.M \in S(C')). \]

Proof

Assume there are sequences \( s \in S(C) \) and \( r \in R(C) \)
\[ \text{s.t. } s \text{ correspondsTo}_C \text{ } r. \]

Following corollary 1 there is also a sequence \( r' \in R(C') \) with \( r' = h \circ r \), i.e.
\[ \forall i : 1..\#r \bullet r'(i) = h(r(i)). \]

Let
\[ s' \equiv s \upharpoonright C'.M. \]

According to the definition of the \((-\text{correspondsTo}-)\)-relation there are three cases to be distinguished.
Case 1:
If \#r' = 1, we have also \#r = 1 and thus s = \{\} = s'.

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

Case 2a
If ((r(1), s(1)), r(2)) \in C'.STD.\delta
then we have either
s(1) \in C'.M
and thus
s(1) = s'(1)
or
s(1) \not\in C'.M
in which case the definition of homomorphism implies:
((r'(1), \epsilon), r'(2)) \in C'.STD.\delta \lor r'(1) = r'(2).

Case 2b
If ((r(1), \epsilon), r(2)) \in C'.STD.\delta \lor r(1) = r(2)
we also have ((r'(1), \epsilon), r'(2)) \in C'.STD.\delta \lor r'(1) = r'(2).

Altogether this leads to

\[
\begin{align*}
\text{s'} &= \{\} \land \#r' = 1 \lor [1] \\
\text{s(1)} &\in C'.M \land ((r'(1), s'(1)), r'(2)) \in C'.STD.\delta \lor [2a] \\
\text{s(1)} &\not\in C'.M \land ((r'(1), \epsilon), r'(2)) \in C'.STD.\delta \lor r'(1) = r'(2). [2b]
\end{align*}
\]

Using

\[s'(2..) \text{ correspondsTo}_{c', r'(2..)}\]
in case 2a and

\[s' \text{ correspondsTo}_{c', r'(2..)}\]
in case 2b as the hypotheses this leads via induction to

\[s' \text{ correspondsTo}_{c', r'}\]
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 inheritance. The notion of homomorphism will be used to formulate sufficient conditions, which imply observability or invocability, respectively.

We shall also give examples of how to construct state transition diagrams for subclasses from those of a superclass in such a way, that the existence of appropriate homomorphisms is guaranteed. 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 we get two different class descriptions $ObsC$ and $InvC$, respectively, for each class. Here $ObsC$ describes the class using the observability approach and $InvC$ describes it according to the invocability approach.

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 5 describes the TV set from the observability point of view. Here additional
\( \epsilon \)-arcs allow "spontaneous" transitions, which are specialized in the class \( TvRemote \).

Figure 5: Observable Behaviour of the TV Set

Let the method identifier sequences described by those two class behaviour descriptions be \( OS(C) = S(ObsC) \) and \( IS(C) = S(InvC) \), respectively. Apparently, as a consistency condition, we have the general assumption that \( IS(C) \subseteq OS(C) \) should hold.
Furthermore, it is even 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. It might be sensible to use exactly one graphical representation for both by graphically distinguishing transitions from $ISTD$ from those transitions, which belong only to $OSTD$.

### 4.2 Observability

In section 2 we introduced the observability approach. To obtain consistency of observability state transition diagrams with inheritance, 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:

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

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:

$$\forall C, C' : Class \mid C \text{ isSubClassOf } C' \bullet$$

$$\exists h : C.STD.Q \rightarrow C'.STD.Q \bullet$$

$$\text{isHomomorphism}(h, C.STD, C'.STD)$$

There are several ways to achieve the existence of such homomorphisms.

The following figures contain some examples of possible transformations on the 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 6).
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 7). 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 8).
In this case the embedding function is an appropriate homomorphism.

4. It is possible to replace \( e \)-transitions in the superclass diagram by transitions labeled with new methods (cf. figure 9).
Then, the identity function has the homomorphism property.

Many more cases, especially combinations of these cases are possible. The notion of homomorphism allows to prove transformations as being behaviour consistent with respect to the observability view.

![Diagram](image)

**Figure 6: Refinement to States**

The observability approach goes hand in hand with the use of \( e \)-transitions. If there are transitions which may be observed with instances of a subclass but which are not invocable by a method, an \( e \)-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
Figure 7: Parallel Extension

Figure 8: Deletion of States and/or Transitions

Figure 9: Replacement of $\epsilon$-Transitions
special case ([EbeEng 94]). Also, \(\epsilon\)-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 inheritance 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:

\[
\forall C, C': \text{Class} \; | \; C \text{ isSubClassOf } C' \; \bullet \\
\exists h : C'.\text{STD}.Q \rightarrow C.\text{STD}.Q \; \bullet \\
isHomomorphism(h, C'.\text{STD}, C.\text{STD})
\]

Then, theorem 2 implies that the existence of a homomorphism implies the desired inclusion.

The following figures contain some example for transformations of the superclass state transition diagram 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 10). The embedding function is the appropriate homomorphism.

2. As a special case the extension by a parallel diagram preserves invocability, too. (cf. figure 11).
3. It is possible to fuse subdiagrams into a single state as long as all transitions are replaced by appropriate loops (cf. figure 12). The function that maps all fused states into the target state is the homomorphism.

Figure 10: Extensions by States and/or Transitions

Figure 11: Parallel Extension
4.4 Summary

If the approach discussed above is used in its full extension, then each class description in the object model will be accompanied by two different state transition diagrams, which lead to different behaviours $OS(C)$ and $IS(C)$. To achieve consistency with inheritance 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.

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 $C'$:

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

$$IS(C') \subseteq IS(C) \subseteq OS(C).$$
5 Related Work

The problem of specializing state transition diagrams in the context of OMT is only shortly mentioned in [RuBiPr 91] in section 5.7.

[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. In addition to our approach, [SaHaJu 94] include 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.

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 dynamic 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]), and others. Since our approach inherently uses states, 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. The formalization and the transformations discussed in the paper may 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.

Acknowledgements We thank the following persons for some helpful discussions during the course of this work: Michael von der Beeck, Gerti Kappel, Gunter Saake, Michael Schrell, Roel Wieringa.
References

|KhoAbn 90| Setrag Khoshafian, Razmik Abnous, *Object Orientation -


