Observable or Invocable Behaviour - You Have to Choose!

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

Gregor Engels
Leiden University
Dept. of Computer Science
P.O. Box 9512
NL-2300 Leiden
phone: ++31-71-27-7096
fax: ++31-71-27-6985
engels@wi.leidenuniv.nl

Abstract

Several object-oriented modeling approaches propose to describe the dynamic behaviour of objects by state transition diagrams. None of them provide 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 ([RuBlPr 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. Especially the interrelation between the behaviour descriptions of a superclass and its subclasses is only superficially described.

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. They deal with a state transition diagram (STD) description of behaviour. *Constructive* approaches like [SaHaJu 94], [LopCos 93], and [McGDye 93] define rules how to modify the STD of a superclass to get a legal STD 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 an STD of a subclass with respect to the STD of a corresponding superclass.

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

- ▷ An STD describes the life cycles of an object, i.e. all *possible sequences* of method calls, which may be invoked on an object.

- ▷ The life cycle description of the subclass has to be compatible with the life cycle description of the superclass. This means that 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 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 STD describes in the sense of a user manual the *executable sequences* of method calls which may be invoked on an object.
- ▷ The life cycle description of the subclass has to be compatible with the life cycle description of the superclass. This means that the life cycle of a subclass may include additional method invocations, but all invocable sequences of method calls of a superclass have 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 in a previous paper [EbeEng 94]. While that paper was restricted to the first kind of modeling behaviour, this paper discusses all 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 in section 3 and section 4 concludes with a discussion.

2 Meaning of Life Cycles

2.1 Observable Behaviour

One way to look at these diagrams 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 calls. 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 the precondition 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) 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 possibility of reaction must also be reflected in the superclass diagram.

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

$$\pi(OS(C)) \subseteq OS(C'),$$

where $\pi(OS(C))$ is the projection of the sequences in $OS(C)$ to the methods of C'.

As an example, we discuss the definition of a class PERSON and of a corresponding subclass EMPLOYEE. 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 EMPLOYEE is constructed by a parallel extension of the superclass state transition diagram. (We use Harel's statechart notation to yield a better readable representation ([Har 87]) than classical state transition diagrams.)

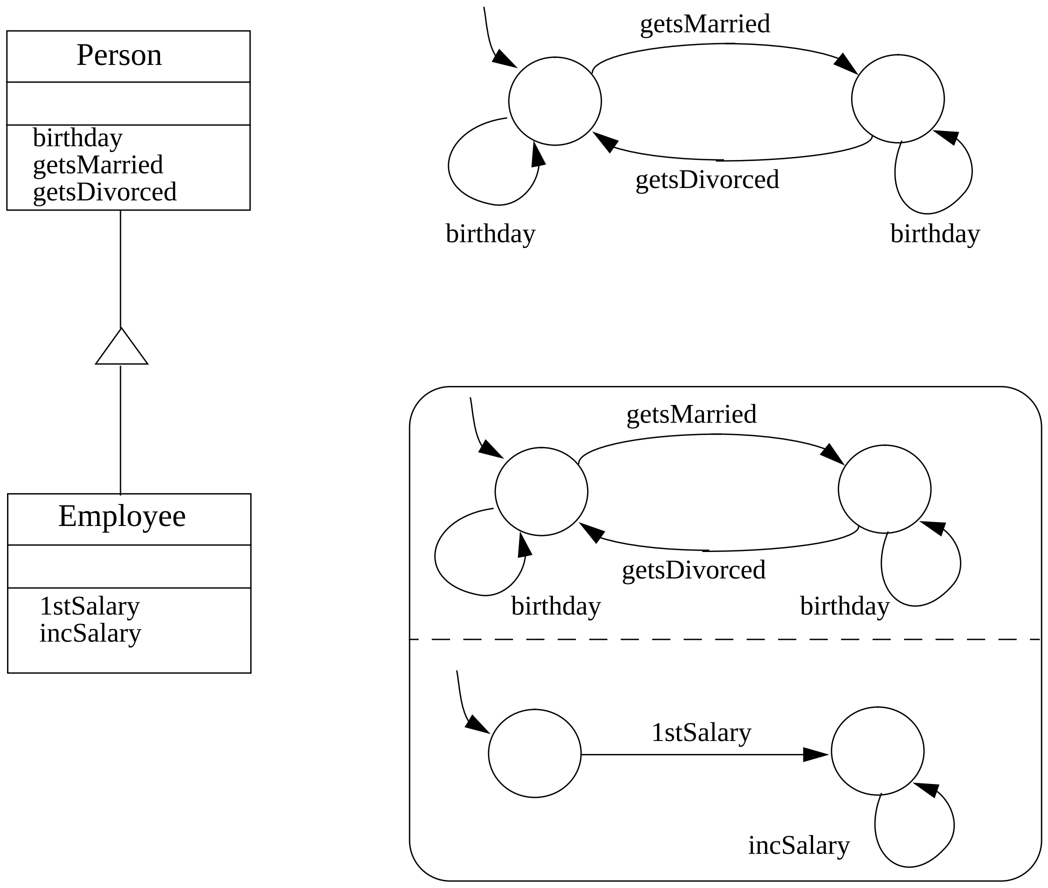

Figure 1: STD of a subclass constructed by parallel extension

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

... birthday 1stSalary getsMarried birthday incSalary birthday ...

A restriction of this life cycle to the methods which are already defined within the superclass PERSON yields a life cycle which is allowed by the state transition diagram of PERSON.

As a second example, we give a subclass definition TRADITIONAL_PERSON, where the corresponding state transition diagram is constructed by refining a state of the state transition diagram of PERSON (cf. Figure 2). Again, the restriction of any life cycle which is observable for an object of TRADITIONAL_PERSON yields a life cycle of a PERSON.

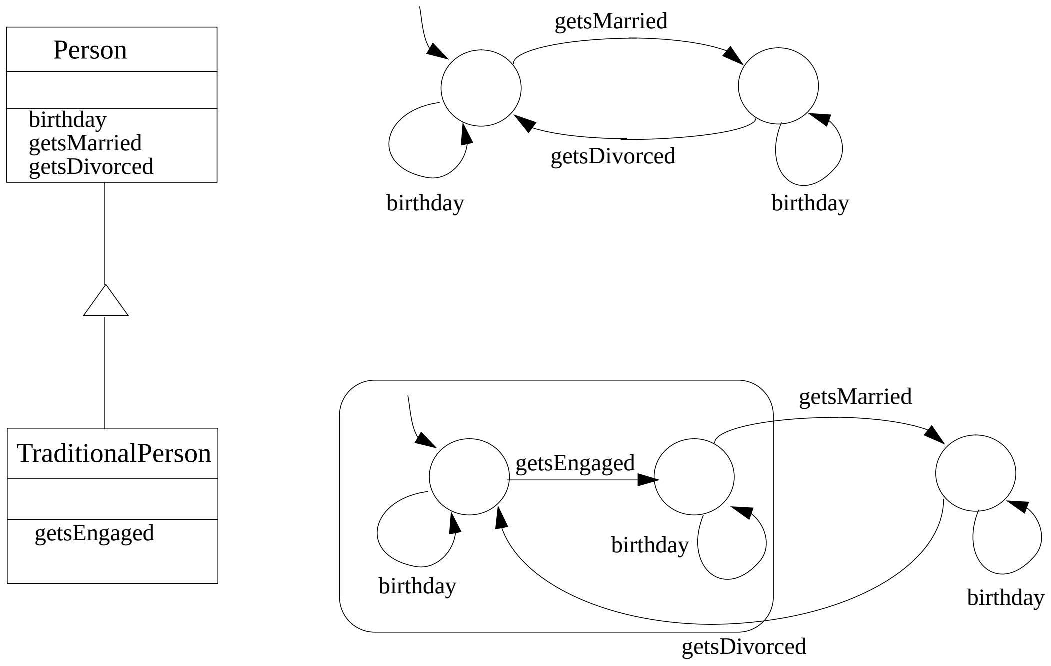

Figure 2: STD of a subclass constructed by state refinement

In this example, the life cycle of a TRADITIONAL_PERSON is much more “restricted” than that for a usual person. To be a TRADITIONAL_PERSON means that one has to fulfill additional prerequisites (“to get engaged”), before one is allowed to get married. Therefore, an (invocable) life cycle of a PERSON is not possible for a TRADITIONAL_PERSON. 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 the precondition of the method is fulfilled. 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 the set of all 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

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

As an example, we discuss the definition of a class TV (television) and of a corresponding subclass TV_RC (television with remote control). Figure 3 gives the class definition as well as the dynamic 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 up-wards or downwards, respectively, and a method "time" which shows the current time for some seconds on the screen. This method "time" may only be invoced at channel 3. The class TV_RC has in addition a remote control with four buttons "ch-i", which enables to choose one of the four channels directly. The

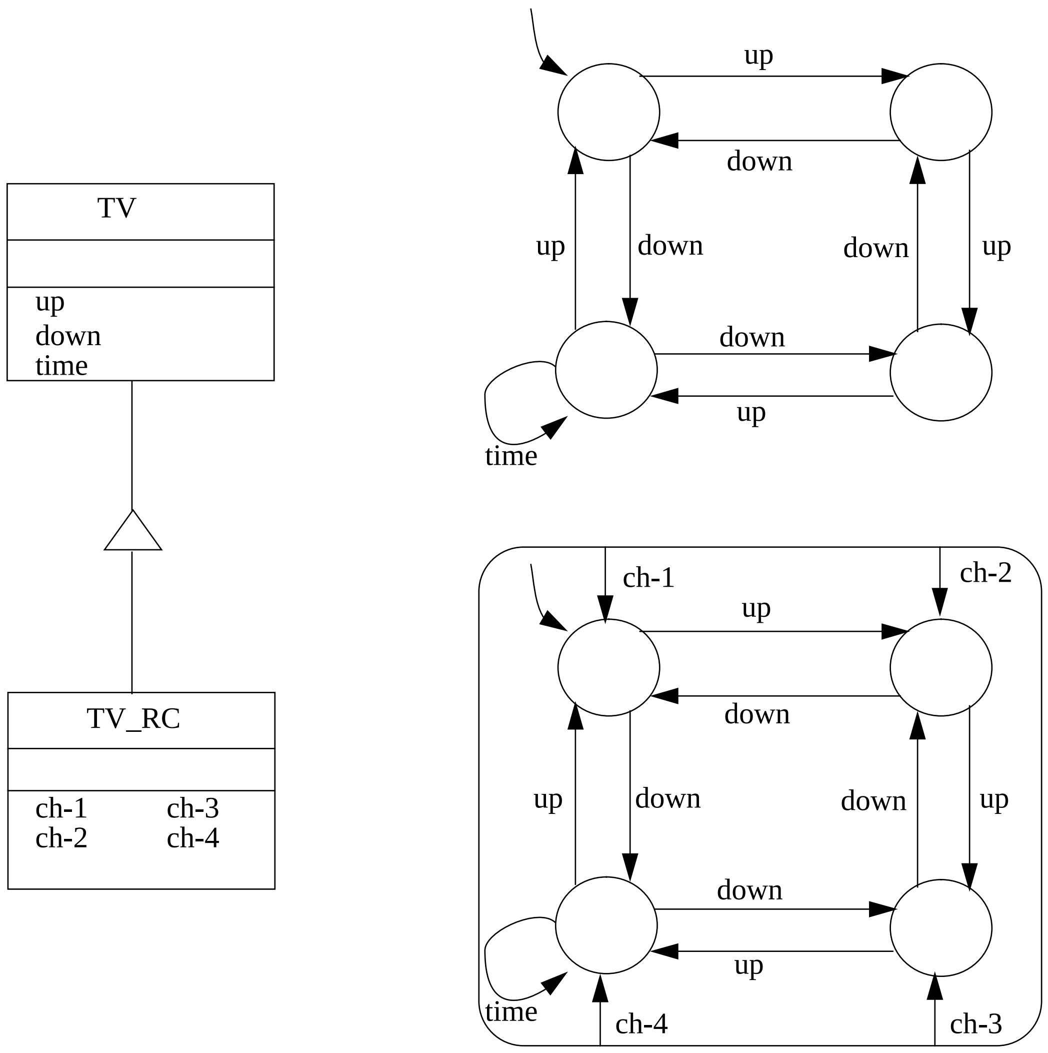

Figure 3: STD of a subclass constructed by adding transitions

statechart of TV_RC describes this situation. It is a short-hand notation for a state transition diagram, where a state representing a channel i has incoming edges labelled by ch-i from all states.

As the state transition diagram of TV_RC is an extension of the state transition diagram of TV, all method sequences invocable on an object of class TV are invocable on an object of class TV_RC. But, in contrast to the above illustrated observable behaviour, the restriction of the following invocable method sequence on an object of class TV_RC

  up ch-3 time up

to the methods already defined in class TV yields the method sequence

  up time up

which is not invocable on an object of class TV.

3 Formalization

To make the concepts described up to now more explicit, we shall give a formal model of the different kinds of integration of the dynamic model in the context of object-oriented modeling.

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 the description.

3.1 Structure

In the following we use a *Z*-like formalization ([Spiv 92]) assuming some universes of identifiers, object identifiers and states.

$$[ID, OID, STATE, VALUE]$$

State transition diagrams (STDs) 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 identifiers that it is able to process, and a relation δ which describes possible state changes together with there possible inputs. Here, the transition relation is non-deterministic and allows spontaneous ϵ-transitions.

STDiagram

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

In order, to describe the integration of the class model and the dynamic model, a rudimentary definition of classes suffices:

Class

$$\begin{array}{ll}
M : \mathbb{F}\, ID & \text{[method identifiers]} \\
STD : STDiagram & \text{[behaviour]} \\
\hline
STD.\Sigma = M &
\end{array}$$

Thus, classes are only described by their method identifiers and their behaviour given by a state transition diagram.

On the set of classes there is a *subclass relationship*, which gives rise to the existence of inheritance. This relationship is usually expressed graphically in the class diagram. In most object-oriented methods it is assumed to be acyclic, and generally subclasses may have more methods than superclasses.

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

At a given point in time there may exist a set of corresponding objects for each class description C. 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 `isInstanceOf` assigning class descriptions to object identifiers.

$$\texttt{isInstanceOf} : OID \nrightarrow Class$$

The domain of `isInstanceOf` 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 is belonging to the extension of) C and all direct or indirect superclasses of C.

$$\texttt{isMemberOf} : OID \leftrightarrow Class$$

$$\texttt{isMemberOf} = \texttt{isInstanceOf};\ \texttt{isSubClassOf}^{*}$$

Thus, objects belong to exactly one class as instances and to several classes as members. The situation of an object with respect to a class that it belongs to is described by a state of the corresponding STD.

3.2 Behaviour

The *behaviour* of an object *oid* is described by the sequence of states that it assumes during its existence. This behaviour has (of course) to be described with respect to a class C of which *oid* is a member. The set of possible steps from one state to another is described by the following `leadsTo`-relation (noted as $\vdash$)

$$\vdash : Class \rightarrow (STATE \leftrightarrow STATE)$$

$$\begin{aligned}
&\forall\, C : Class;\ q, \widehat{q} : STATE \bullet \\
&\quad q \vdash_C \widehat{q} \\
&\quad \Leftrightarrow \\
&\qquad \{q, \widehat{q}\} \subseteq C.STD.Q \wedge \\
&\qquad\quad (q = \widehat{q} \vee \\
&\qquad\quad \exists\, m : C.M \cup \{\epsilon\} \bullet ((q, m), \widehat{q}) \in C.STD.\delta)
\end{aligned}$$

The situations that an object *oid* may be in and the temporal steps (given by $\vdash$) that an *oid* might perform must of course fulfill some compatibility restrictions with respect to all classes that *oid* is a member of. The kind of restrictions, that one may pose on them, depends on the understanding of the meaning of the STD, as described in section 2.

3.3 Observable Behaviour

Let *oid* be a member of class C and assume that C is a subclass of class C'. The *observability approach* described above implies that each situation with respect to C must be mapped onto a situation with repect 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 $h : STD.Q \to STD'.Q$ is a homomorphism if the following condition is fulfilled:

$$isHomomorphism : (STATE \to STATE) \leftrightarrow STDiagram \leftrightarrow STDiagram$$

$$\begin{array}{l}
\forall\, h : STATE \to STATE;\ STD, STD' : STDiagram \bullet \\
\quad isHomomorphism(h, STD, STD') \\
\quad \Leftrightarrow \\
\qquad h \in STD.Q \to STD'.Q \\
\qquad \forall\, q, \widehat{q} : STD.Q;\ m : STD.\Sigma \cup \{\epsilon\} \bullet \\
\qquad\quad ((q, m), \widehat{q}) \in STD.\delta \\
\qquad\quad \Rightarrow \\
\qquad\qquad ((h(q), m), h(\widehat{q})) \in STD'.\delta \vee \\
\qquad\qquad ((h(q), \epsilon), h(\widehat{q})) \in STD'.\delta \vee \\
\qquad\qquad h(q) = h(\widehat{q}) \\
\qquad h(STD.q_0) = STD'.q_0
\end{array}$$

Using this definition, we can show that each legal behaviour at the level of a subclass C is also observable as a legal behaviour at the level of its superclass C':

Theorem

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

3.4 Invocable Behaviour

On the other hand, using the identifiers from above, the *invocability approach* implies that 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 also executable with respect to C.

To achieve this we require an *embedding* k of the state transition diagram of C' into that of C. If STD' and STD are state transition diagrams an embedding of STD' into STD is a function $k : \mathit{STD}'.Q \rightarrow \mathit{STD}.Q$ which fulfills the following condition:

$$
\begin{array}{|l}
\mathit{isEmbedding} : (\mathit{STATE} \rightarrow \mathit{STATE}) \leftrightarrow \mathit{STDiagram} \leftrightarrow \mathit{STDiagram} \\
\hline
\forall\, k : \mathit{STATE} \rightarrow \mathit{STATE};\ \mathit{STD}', \mathit{STD} : \mathit{STDiagram} \bullet \\
\quad \mathit{isEmbedding}(k, \mathit{STD}', \mathit{STD}) \\
\quad \Leftrightarrow \\
\qquad k \in \mathit{STD}'.Q \rightarrow \mathit{STD}.Q \\
\qquad \forall\, q', \widehat{q'} : \mathit{STD}'.Q;\ m : \mathit{STD}'.\Sigma \cup \{\epsilon\} \bullet \\
\qquad\quad ((q', m), \widehat{q'}) \in \mathit{STD}'.\delta \\
\qquad\quad \Rightarrow \\
\qquad\qquad ((k(q'), m), k(\widehat{q'})) \in \mathit{STD}.\delta \\
\qquad k(\mathit{STD}'.q_0) = \mathit{STD}.q_0
\end{array}
$$

Using this definition, we can show that each invocable behaviour at the level of a superclass C' is also an invocable behaviour for its subclass C:

Theorem

$$\begin{array}{l}
\forall\, C, C' : \mathit{Class};\ k : \mathit{STATE} \rightarrow \mathit{STATE} \bullet \\
\quad C\ \texttt{isSubClassOf}\ C' \wedge \\
\quad \mathit{isEmbedding}(k, C'.STD, C.STD) \\
\quad \Rightarrow \\
\qquad \forall\, q', \widehat{q'} : C'.STD.Q \bullet \\
\qquad\quad q' \vdash_{C'} \widehat{q'} \Rightarrow k(q') \vdash_{C} k(\widehat{q'})
\end{array}$$

4 Conclusion

Putting both results together we come to the following alternative:

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

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

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

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

Section 2 gave concrete examples for these two approaches. Abstracting from them, one may derive guidelines how an STD of a class may be modified in order to get an STD of a subclass. Figure 4 shows two possibilities how an STD may be modified according to the observability approach, namely a modification by parallel extension or by state refinement[1]. All labels of newly introduced transitions have to be method identifiers added within the

[1]These examples correspond directly to cases C and D of the examples given in [McGDye 93].

subclass. Note that it is not necessary that all refined substates are connected with the old environment. For instance, there is no transition between state *d* and state *b* via transition *f*, while there is one between state *e* and state *b* in the right subclass STD of figure 4.

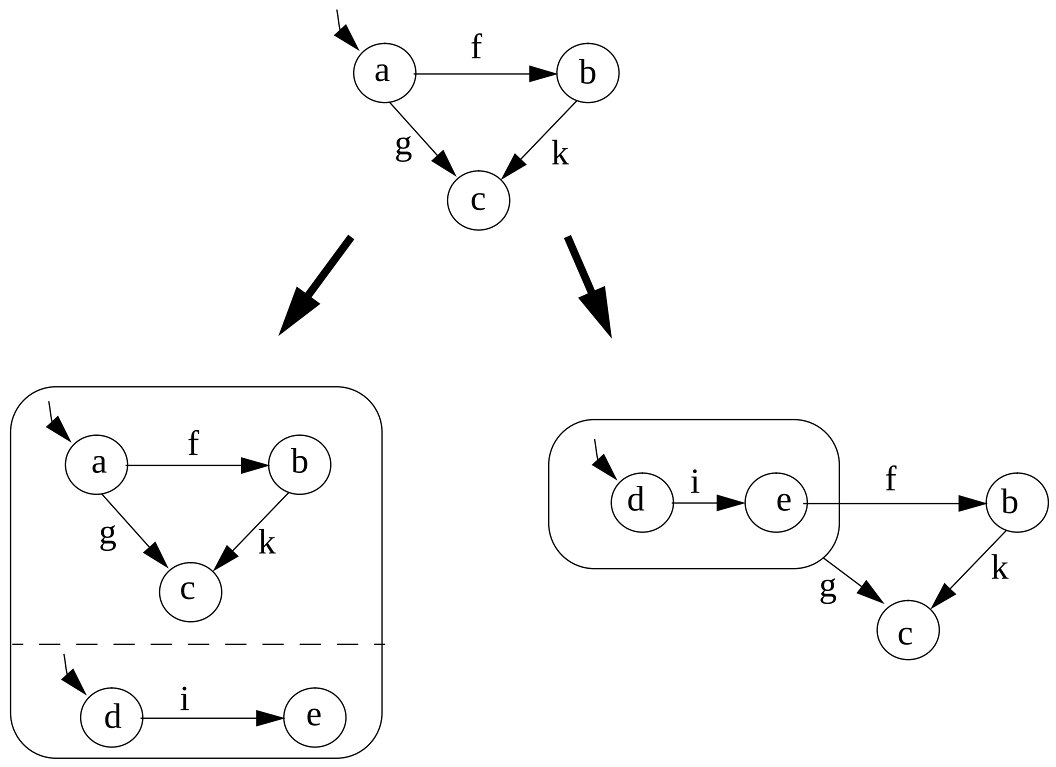

Figure 4: Modifications within the observability approach

In the left case, an appropriate homomorphism *h* is a projection of each composed state of the lower STD to its upper component. In the right case, a homomorphism *h* maps all refining states of the lower STD to the refined state of the upper STD leaving all other states unchanged.

Figure 5 shows two possibilities how an STD may be modified according to the invocability approach. In the left case, the old STD is extended by an additional transition [2]. In the right case, the old STD is even extended by a

[2]This corresponds to case B in [McGDye 93].

subdiagram, which is connected to some of the old states. In both cases, the identity function can be chosen as the embedding function k.

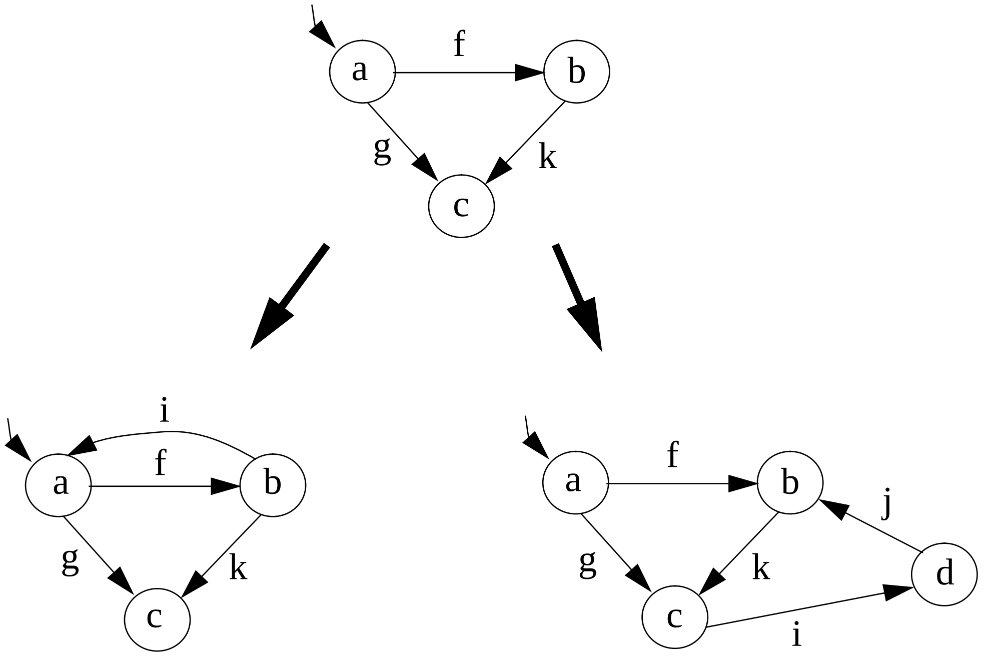

Figure 5: Modifications within the invocability approach

Let, again, $OS(C)$ and $IS(C)$ be the set of all observable respectively invocable sequences of method calls to a class C. Since we have $IS(C) \subseteq OS(C)$ we get for the method sequences of a a subclass C and a superclass C' the following inclusions:

$IS(C') \subseteq \pi(OS(C)) \subseteq OS(C')$
and
$IS(C') \subseteq IS(C) \subseteq OS(C)$.

There is a strong connection between the different uses of state transition diagrams in the context of object-oriented modeling. Thus, there has a definite *conceptual choice* to be done about which approach is to be used when building integrating tools for the support of object-oriented analysis and design methods.

References

[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.

[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*, TU 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 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.

[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.

[RuBlPr 91] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs NJ, 1991.

[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.

[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.