Equivalence Transformations on Statecharts

Heinz Frank and Johann Eder
Universität Klagenfurt, Institut für Informatik-Systeme
E-mail: {heinz, eder}@ifi.uni-klu.ac.at

Abstract

Statecharts are a popular representation technique for conceptual models of the dynamics of a universe of discourse. However, designers are not supported in their work with dynamic models as well as they are for working with static models. We present a meta-model and a formalization of the semantics of a statechart language. Important results are the definition of the equivalence of statecharts and a sound and complete axiomatization of the equivalence. Based on this we define a set of basic schema transformations which do not change the semantics of a model. These transformations can be used to successively transform statecharts to achieve design goals or to prepare them for implementation.

1. Introduction

Conceptual modeling of a universe of discourse has two dimensions: the structure of objects and their relationships are represented in a static model (or object model) and the behavior of the objects is documented in a dynamic model ([15, 2, 16, 3]). While the techniques for structural modeling have a long tradition and are already quite elaborated, conceptual modeling techniques for the dynamics of a mini-world is not supported as well. Open issues are for example the formalization of the semantics of dynamic models, generalization and inheritance of dynamic models ([12, 4, 17]), and transformations of dynamic models. The aim of the work reported here is to contribute to a better understanding of dynamic models and to support the modeling process.

Statecharts, introduced by D. Harel ([9, 10, 11]) are a popular method for designing the behaviour of objects. This concept is used in various design methodologies e. g. OMT ([16]), OOD ([2]) or UML ([15]). Since the introduction of statecharts many variants of statecharts have been proposed. Von der Beeck ([18]) provides a detailed analysis of around 20 statecharts variants.

For designing static models, designers or analysts start from an initial model and successively transform this model to achieve design goals and meet quality criteria. In the end, the model is in a form which is well suited for mapping to a logical model and thus serves as a specification of the implementation. This process is supported by a well understood representation language and the provision of schema transformations which maintain the semantics of the model (i. g. [1]). In our opinion, a similar process should be made available for the development of dynamic models.

Assumptions and scope

For this work we assume that the static part of the model is already developed. For the representation of dynamic aspects we focus on the modeling of the dynamics of a single type or class of the static model. We represent dynamic models with statecharts. We consider these statecharts to serve several purposes. First, they are a representation technique to capture the dynamics of objects in the universe of discourse. Second, statecharts support the communication between users, analysts, designers and implementors. Finally, statecharts are (partial) specifications for the implementation of an information system. In this paper we focus more on the formal aspects of statecharts, i.e. their semantics as partial specification of methods and not on the pragmatics and style which is of great importance for communication purposes.

The major contributions of this paper are

- formalization of statecharts for conceptual modeling
- definition of the semantics of statecharts as partial method specification
- definition of the equivalence of statecharts together with a sound and complete axiomatization
- a complete set of basic schema transformations for deriving equivalent statecharts

The paper is organized as follows. In section 2 we introduce the statechart language and a meta-model for statecharts. The major concepts of statecharts are demonstrated with an example from the domain of a library. In section 3 we discuss the equivalence of statecharts as equivalence of method specifications. In section 4 we present a set of basic equivalence transformations for statecharts. In section 5 we draw some conclusions and discuss some applications of this work. Space limitations force us to omit several proofs in this paper. Interested readers are referred to [6] where proofs for all the theorems in this paper are provided.

2. The statechart language

For the structural part of types we are using a very simple data model (according to [5]). A type is a labeled set of type attributes. E. g. type book = [title: string, pages: integer] is a type where *book* is name of the type. *Title* and *pages* are the attributes of the type. Attributes are typed with basic types, such as *string* or using labels of user defined types.

For the representation of the behaviour of a type we are using a statechart language ([9, 10, 11]). A statechart of a type primarily consists of *states*, *events* and *transitions*. The major elements of statecharts are shown in figure 1.

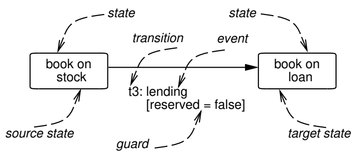

Figure 1. The basic elements of statecharts

A *state* is a condition or situation in the life time of an object during which it satisfies some condition. An object satisfying the condition of a state, is said to be in that state ([15]). We call this condition the *range* of a state.

A *transition* is a kind of relationship between two states and is triggered by an *event*. A transition indicates that an object which is in the first state (called *source state*) will enter the second state (called *target state*) when the event occurs and some specified condition (called the *guard* of the transition) holds ([15]). At the end of a transition the object is in the target state of the transition.

The Library

We choose the domain of a library as an example. Suppose the static model contains a type book with some attributes, such as title, isbn, signatures . . .

The behaviour of books is shown in figure 2. Two major steps are necessary for the administration of books before they can be placed into the library, the book registration and the book preparation. The book registration is responsible for recording new books. For this purpose the reference to the book catalogue must be known. In the next step a signature is given to the book. The book preparation first describes a book with the subject and afterwards with some keywords.

After finishing the administration books are placed into the library, where they can be borrowed. Books, which are

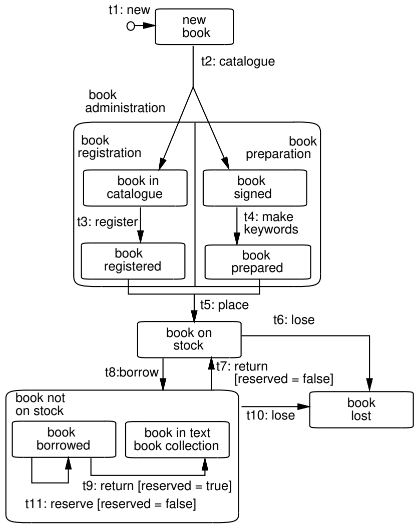

Figure 2. The behaviour of a book

necessary for a lecture are given in a special place called text book collection. Nobody is allowed to borrow books from the text book collection. If a book should be placed into the text book collection but is borrowed by anyone else it can be reserved. Reserved books may not be borrowed by anyone but are placed into the text book collection immediately after they are returned by the borrower.

Books, which are out of stock (i. e. books which are borrowed or in the text book collection) are returned to the library, if they are not reserved. We also consider that books might get lost.

In the next sections the example and the meta model (shown in figure 3) will be used to discuss the formalization of the statechart language more detailed.

2.1. States

The static model spans an object space, which is defined as the set of all possible extensions of the static model. A state in a statechart of a type T is a subset of possible object instances of T, it is a subspace of the object space of the type. Intensionally, a state is defined by a predicate for objects of the given type. Extensionally, a state is considered as the set of all possible objects which fulfill this predicate.

To make statecharts more readable and to avoid combinatorical explosion of nodes and arcs, state hierarchies have been introduced. According to Harels definition ([11]) we distinguish between *OR-states*, *AND-states* and *basic states*. OR-states have substates which are related to each other by “exclusive-or”, i. e. an object of a type can only be in one substate of an OR-state at any time. AND-states have orthogonal components which are “and” related. An object, that is in an AND-state is also in all substates of the

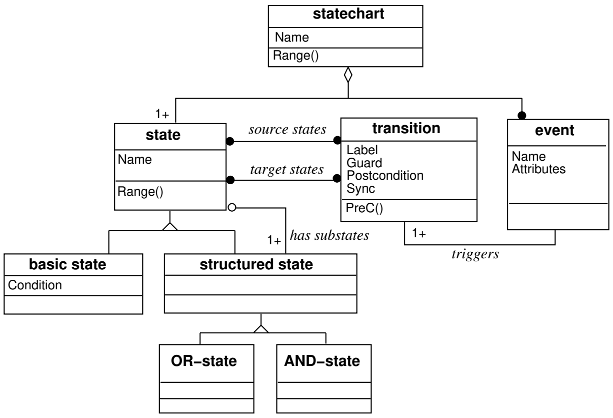

Figure 3. A meta-model for statecharts

AND-state. Basic states are the states at the bottom of a state hierarchy, they do not have substates. The states at the highest level of a statechart, that are states without a parent state, are called *root states*.

We use $\mathcal{TQL}++$ ([13],[14]) as specification language for these conditions (the range of states, pre- and postconditions and guards of transitions). This language allows the definition of logical conditions, which objects have to satisfy. E. g. the range of the state solvent of a type bank in $\mathcal{TQL}++$ would be $this.assets > 0$. To be an object of this type in this state the value of its attribute $assets$ must be greater than zero.

The notation $S.Range()$ is used for denoting the range of the state S. While the range of basic states must be given by the designer, the range of the structured states (AND-states and OR-states) is computed.

Definition 2.1 The range of a state S is defined as

- $S.Condition$, if S is a basic state.
- the disjunction of the ranges of all the substates of S, if S is an OR-state.
- the conjunction of the ranges of all the substates of S, if S is an AND-state. □

Example (Fig. 2): Some of the conditions of the basic states of the library example are shown in table 1. The range of the OR-state book not on stock is defined as the disjunction of its substates book borrowed and book in text book collection. The range of the AND-state book administration is definded as the conjunction of its substates book registration and book prepararation, which are OR-states.

The ranges of states allow the definition of relationship between states. These relationships are used for the definition of the correctness of states according to Harels definition ([11]) in a more formal way.

Definition 2.2 Let $Z(o)$ be a predicate returning $true$ if the object o satisfies the range of the state Z. Let $P(T)$ be the set of all possible instances of the type T. Then two states Z_1 and Z_2 of the statechart of the type T are called

equivalent, if	$\forall o \in P(T) : Z_1(o) \leftrightarrow Z_2(o)$
orthogonal, if	Z_1, Z_2 are OR-states with equivalent ranges and $\forall z \in Z_1.Substates$, $\forall z' \in Z_2.Substates \rightarrow \exists o \in P(T) : z(o) \wedge z'(o)$
disjoint, if	$\forall o \in P(T) : \neg (Z_1(o) \wedge Z_2(o))$ □

Two OR-states Z_1 and Z_2 are *orthogonal* if their ranges are equivalent and each substate z of Z_1 overlaps with each substate z' of Z_2 (and vice versa).

book on stock	this.position = in library
book borrowed	this.position = borrowed $\wedge$ (this.reserved = true $\vee$ this.reserved = false)
book in text book collection	this.position = in text book collection $\wedge$ this.reserved = false
book lost	this.position = lost

Table 1. Conditions of the basic states

Definition 2.3 The states of a statechart are *correct*, if

1. all root states are disjoint,
2. all substates of an OR-state are disjoint
3. all substates of an AND-state are orthogonal. □

Example (Fig. 2): The substates of the OR-state book not on stock must have disjoint ranges, i. e. no object can satisfy both conditions at the same time (compare table 1). The substates of the AND-state book administration must be OR-states with equivalent ranges. Furthermore, it must be possible for an object to satisfy e. g. the ranges of the states book registered and book signed at the same time.

As a consequence of definition 2.3 the substates of an AND-state must be OR-states, as all substates of an AND-state must be orthogonal (definition 2.2). Therefore, basic states can only be direct substates of an OR-state.

2.2. Events and Transitions

An event is an incident directed to an object with the aim to change the state of the object. An event is set off explicitly and triggers a transition which changes the state of an object. In dynamic modeling events and transitions represent (partial) specifications of the methods for the object type. If an event is set off, an object is transfered to a new state. The model defines which conditions (preconditions) an object has to fulfill in order to be able to react to an event and which conditions (postconditions) an object satisfies after the state change.

Sync defines whether the transition is *synchronizing*. Synchronizing transitions always lead from and/or to a state

t6: lose	this.position = lost
t8: borrow	(this.position = borrowed ∧ this.reserved = false) ∨ (this.position = in text book collection ∧ this.reserved = false)
t10: lose	this.position = lost
t11: reserve	this.position = borrowed ∧ this.reserved = true

Table 2. The postconditions of the transitions

aggregation (as for instance the transition t2 triggered by the event catalogue). Synchronizing transition may have more than one source (or target) state.

To cause a state change the object must be in the source state of the transition (that means satisfying the range of the source state) and the object must satisfy the condition of the guard of the transition. Therefore, the precondition of a transition is defined as the conjunction of the range of the source states and the guard of the transition. We use $t.PreC()$ for the precondition of a transition t.

After the application of a transition the object satisfies the range of its target state and its postcondition. Therefore, the postcondition of a transition must imply the range of its target state.

Example (Fig. 2): The precondition of the transition t9 is defined as the conjunction of the range of the state book borrowed and its guard this.reserved = true resulting in this.position = borrowed ∧ this.reserved = true. Some of the postconditions, given by the designer, are shown in table 2

Definition 2.4 Transitions are correct, it they have source and target states according to the following conditions:

1. Non-synchronizing transitions triggered by transforming events have exactly one source and one target state.
2. Synchronizing transitions could have several source and target states. If there are several source states, they must be part of the same state aggregation. If there are several target states, they must be part of the same state aggregation too.
3. Transitions triggered by object producing events do not have source states.
4. Transitions triggered by object destroying events do not have target states, their postcondition must be $true$.
5. The postcondition of a transitions has to imply the range of each of its target states. □

2.3. Correct Statecharts

Statecharts have a range too, which is defined as:

Definition 2.5 The *range* of a statechart is the disjunction of the ranges of all *root* states of the statechart. □

Based upon the ranges of states and the correctness of transitions and states we define a correct statechart. In our following considerations we assume correct statecharts.

Definition 2.6 A statechart is called *correct* if (1) all states are correct according to definition 2.3, and (2) all transitions are correct according to definition 2.4. □

3. Equivalence of dynamic models

We would like to support designers to work with statecharts in a similar way as they already can do with static models. In particular, our goal is to support the transformation of statecharts without changing the semantics. For this purpose we need a clear definition when statecharts are equivalent. Our definition is based on the consideration that statecharts are equivalent, if they provide the same partial specification for the development of methods. Therefore, the equivalence of correct statecharts ($M_1 \equiv M_2$) bases on equivalent model ranges and equivalent events. We will first define the equivalence in a model-theoretic way and then present a sound and complete axiomatization which will then be used to prove that schema transformations preserve the semantics of the schema.

The *specification* of an event is a set of condition pairs of the form $\{(Pre_1, Post_1), \ldots, (Pre_n, Post_n)\}$. One pair $(Pre_i, Post_i)$ indicates that an object which satisfies the condition Pre_i (we say $Pre_i(o)$ is true, if the object o satisfies the condition Pre_i) after the application of the event (actually of the corresponding transition triggered by the event) satisfies the condition $Post_i$. We take the pre- and postconditions of the transitions triggered by the event e in order to calculate the specification of the event:

Definition 3.1 The specification of the event e is defined as $e.Spec = \{(t.PreC(), t.Postcondition) \mid t \in e.triggers\}$ □

The specification of an event is computed by collecting all the pre- and postconditions of the corresponding transitions of the event (listed in *e.triggers*). In our consideratons the specification of an event e is named $e.Spec()$.

Example (Fig. 2): The specification of the event lose of figure 2, p. 2, is defined as: lose.Spec() ≡ {(t6.PreC(), t6.Postcondition),(t10.PreC(), t10.Postcondition)}

The *semantics* of a statechart M is defined as the range of the statechart and the set of its event specifications.

Definition 3.2 The semantics of a statechart M is defined as $(M.Range(), \{(e_i, e_i.Spec()) \mid e_i \in M.Events\})$. □

Example (Fig. 2): The semantics of the statechart M_1 of figure 2, p. 2, is defined as (M_1.Range(), {(new, new.Spec()), (lose, lose.Spec()), …}), which is (M_1.Range(), {(new, {(t1.PreC(), t1.Postcondition)}), (lose, {(t6.PreC(), t6.Postcondition), (t10.PreC(), t10.Postcondition}), …}).

Definition 3.3 The predicate $e.Post(o)$ of an event e and an object o is defined as $e.Post(o) := \bigvee\{Post|\exists Pre : (Pre, Post) \in e.Spec() \wedge Pre(o)\}$ whereby $\bigvee\emptyset = false$, $\bigvee\{Post\} = Post$ and $\bigvee\{Post_1 \ldots Post_n\} = Post_1 \vee \ldots \vee Post_n$ □

The predicate $Post(o)$ is defined as the disjunction of all postconditions of conditional pairs $(Pre, Post)$ from $e.Spec()$ for which the object o satisfies the precondition $(Pre(o))$. $E.Post(o)$ is $false$, if the object o doesn't satisfy any of the preconditions of the event specification of e.

Now two event specifications are equivalent, if $Post$ applied to both specifications for all objects in $P(T)$ returns equivalent conditions. In other words two events are equivalent, if each possible object satisfies the same condition after the events occurred.

Definition 3.4 The event specifications of the two events e_1 and e_2 are *equivalent* ($e_1.Spec() \equiv e_2.Spec()$), if $\forall o \in P(T) : e_1.Post(o) \leftrightarrow e_2.Post(o)$ □

As there might be several (different) events with equivalent specifications we have to take the name of the events into account when we define the equivalence of events.

Definition 3.5 Two events e_1 and e_2 are *equivalent* ($e_1 \equiv e_2$), if they have the same name and equivalent event specifications. □

The following theorem states that the equivalence of events is well defined, i. e. it is an equivalence relation.

Theorem 3.1 The equivalence $\equiv$ of events is *reflexive*, *symmetrical* and *transitive*, hence an equivalence relation. □

We define the equivalence of statecharts ($M_1 \equiv M_2$) based upon the range of the statecharts and the equivalence of their events, and show, that it is well defined (i. e. it is an equivalence relation).

Definition 3.6 Two correct statecharts M_1, M_2 are *equivalent* ($M_1 \equiv M_2$), if their ranges are equivalent and for all events of M_1, there exists an equivalent event in M_2, and for all events of M_2, there is an equivalent event in M_1. □

Theorem 3.2 The equivalence of correct statecharts is *reflexive*, *symmetrical* and *transitive*, hence an equivalence relation. □

To examine whether two statecharts are equivalent according to this definition seems to be quite cumbersome. We would like to have a set of transformation on event specifications which are easier to apply. We are interested which changes of event specifications are possible without leaving an equivalence class.

We define the relation Ξ (say:derive) for event specifications. It means the left part of the relation Ξ can be changed to the right part and vice versa.

Definition 3.7 Let S, S_1, S_2 and S_3 be event specifications. Let additionally Pre, Pre_1, Pre_2, Pre_i and Pre_j as well as $Post$, $Post_1$, $Post_2$, $Post_i$ and $Post_j$ be Pre- and Postconditions ($\mathcal{TQL}++$ terms). Then:

(1) $S \cup \{(Pre_1, Post), (Pre_2, Post)\} \; \Xi_1$
$S \cup \{(Pre_1 \vee Pre_2, Post)\}$

(2) $S \cup \{(Pre, Post_1), (Pre, Post_2)\} \; \Xi_1$
$S \cup \{(Pre, Post_1 \vee Post_2)\}$

(3) $\{(false, Post)\} \; \Xi_1 \; \emptyset$

(4) $\{(Pre, false)\} \; \Xi_1 \; \emptyset$

(5) $\{(Pre_i, Post_i)\} \; \Xi_1 \; \{(Pre_j, Post_j)\}$
if $Pre_i \leftrightarrow Pre_j \wedge Post_i \leftrightarrow Post_j$

(6) $(S_1 \; \Xi \; S_2) \wedge (S_2 \; \Xi_1 \; S_3) \; \rightarrow \; S_1 \; \Xi \; S_3$ □

According to the definitions 3.7(1) and (2) we may summarize event specifications with equivalent postconditions through disjunction of their preconditions as well as event specifications with equivalent preconditions through disjunction of their postconditions. The definitions 3.7(3) and (4) allow us to remove event specifications whose pre- or postconditions result in $false$. The definition 3.7(5) states that pairs of event specifications following to the relation Ξ are equivalent if their pre- and postconditions are equivalent terms (in our work equivalent $\mathcal{TQL}++$ terms). In definition 3.7(6) the transitivity of the relation Ξ is determined.

Example (Fig. 2): Consider the specification of the event lose which are based on the transitions t6 and t10 (shown on page 4). As the postconditions of the transitions are equivalent (compare table 2, p. 4) we may combine the specification by the disjunction of the preconditions:
{(t6.PreC(), t6.Postcondition), (t10.PreC(), t10.Postcondition)} Ξ {(t6.PreC() $\vee$ t10.PreC(), t10.Postcondition)}

Theorem 3.3 Let S_1, S_2, T be event specifications. Then (1) $S_1 \; \Xi \; S_1$ and (2) $S_1 \; \Xi \; S_2 \rightarrow (S_1 \cup T) \; \Xi \; (S_2 \cup T)$ □

We define that a statechart M_2 is derived from a statechart M_1 ($M_1 \; \Xi \; M_2$), if all event specifications of M_2 are derived from M_1 and vice versa.

Definition 3.8 Let M_1 and M_2 be statecharts. $M_1 \; \Xi \; M2$ if the ranges of M_1 and M_2 are equivalent and for all events e of M_1 there is an event e' in M_2 with $e \; \Xi \; e'$ and for all events e of M_2 there is an event e' in M_1 with $e \; \Xi \; e'$ □

The operations, defined by the relation Ξ were developed in order to express the same relation as $\equiv$. Therefore, Ξ is a sound and complete axiomatization of the equivalence relation for event specification. This is expressed in the following theorems (for the proofs we refer to [6]).

Theorem 3.4 Let S_1 and S_2 be event specifications. From $S_1 \; \Xi \; S_2$ follows $S_1 \equiv S_2$. □

Theorem 3.5 Let S_1 and S_2 be event specifications. From $S_1 \equiv S_2$ follows $S_1 \Xi S_2$. $\square$

An immediate consequence of these theorems is that the relation Ξ is a sound and complete axiomatization of the equivalence of statecharts too.

Theorem 3.6 Let M_1 and M_2 be correct statecharts. Then $M_1 \equiv M_2 \leftrightarrow M_1 \Xi M2$. $\square$

Now we are able to check whether two statecharts are equivalent by checking if the event specifications of one statechart can be derived from the event specifications of the other statechart. Furthermore, the definitions and theorems in this section provide a formal basis for discussing equivalence transformations of statecharts.

4. Schema transformations

Schema transformations are operations on a statechart M_1 resulting in a different statechart M_2. Each schema transformation deals with a certain aspect of the statechart (e. g combines states or shifts transitions within a state hierarchie). In the following we present a set of 23 basic schema transformations which do not change the semantics of the statechart according to the definition of equivalence given above. Due to the transitivity of the equivalence of dynamic models complex transformations can be established on this basic set of transformations. In our approach the transformations are treated as meta-methods of the meta-model (e. g. as meta-methods for states). Due to space limitations we are not able to discuss them all in detail and omit the formal proofs. Again interested readers are refered to [6].

4.1. Shifting transitions within a state generalization

(1) **UpSg**(Z) shifts a transition with the source state Z to the parent state of Z. The parent state of Z must be an OR-state

The source state Z of the transition is replaced by the parent state of Z. However, as the parent state of Z is an OR-state with a "wider" range than the substate Z, the guard of the transition must replaced by the precondition of the transition to guarantee that the transition can only be applied to objects that comply with the original precondition.

Example: Suppose we would like to shift the transition t9 to the OR-state (figure 2, p. 2) . t9.UpSg(book borrowed) changes the source state of the transition to book not on stock. The guard of the transition must be changed to book borrowed.Range() $\wedge$ reserved = true. The result of the transformation is shown in figure 4(a)

(2) **UpTg**(Z) shifts a transition with the target state Z to the parent state of Z. The parent state of Z must be an OR-state.

The target states Z of the transition is replaced by the superstate of Z. Pre- and Postconditions of the transitions remain unchanged.

Example: t9.UpTg(book in text book collection) changes the target state of the transition to the OR-state book not on stock.

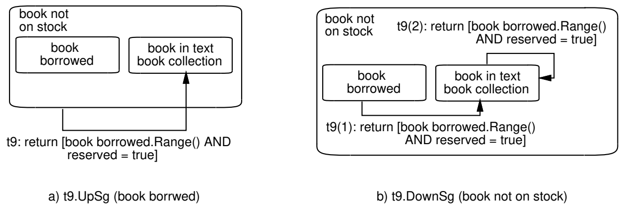

Figure 4. Shifting a transition

(3) **DownSg**(Z) shifts a transition having the OR-state Z as source state to all substates of Z.

For each substate of Z the transition must be copied and the source states are adopted. The original transition is deleted. Afterwards some of the new transitions may have preconditions resulting in $false$. However, according to definition 3.7, such transitions can be deleted. Later we present the schema transformation $Clean$ for that purpose.

Example: Consider the example in the previous part where we shifted the transition t9 to the OR-state using t9.UpSg(book borrowed) (figure 4(a)). Now we would like to shift the transition back to its original source state using the transformation t9.DownSg (book not on stock). For each substate the transitions is copied and the source states are changed. The result is shown shown in figure 4(b).
Let's analyze the preconditions of the new transitions. t9(1).PreC() results in the conjunction of the range of the source states and the guard, that is book borrowed.Range() $\wedge$ book borrowed.Range() $\wedge$ reserved = true which is obviously equivalent to the precondition of the transition from the example in figure 2. The precondition of t9(2) results in book in text book collection.Range() $\wedge$ book borrowed.Range() $\wedge$ reserved = true. However, as in a correct statechart the ranges of the substates of an OR-state must be disjoint (compare definition 2.6), this precondition results in $false$. The transition t9(2) can be deleted (compare definition 3.7).

(4) **DownTg**(Z) shifts a transition having the OR-state Z as target state to all substates of Z.

For each substate of Z the transition must be copied and the target states adopted. The postconditions of a copied transition must replaced by the conjunction of the original postcondition and the range of its new target state. If afterwards the postcondition of a transition results in $false$, it can be deleted (compare definition 3.7). The original transition is deleted

4.2. Shifting transitions within a state aggregation

(5) **UpSa**(Z) shifts a transition having Z as source state to the parent state of Z. The parent state of Z must be an AND-state.

(6) **UpTa**(Z) shifts a transition having Z as target state to the parent state of Z.The parent state of Z must be an AND-state.

Both transformations are very simple, the source (target) state Z of the transition is replaced by the parent state of Z. According to the definition of a correct statechart (definition 2.6) the substates of an AND-state must be OR-states with equivalent ranges. Due to the definition 2.1 the ranges of the AND-state and its substates are equivalent too.

However, after shifting a synchronizing transition to an AND-state the AND-state may appear several times in the source (target) states of the transition. Such redundant stored source or target states can be removed.

Example: We shift the transition t5 from figure 2 first to the OR-state (t5.UpSg(book registered)) and t5.UpSg(book in subject catalogue)) and then to the AND-state (transformations t5.UpSa(book registration) and t5.UpSa(book preparation)). Afterwards the AND-state book administration appears twice in the source states of t5. We remove one and change t5 to a non-synchronizing transition.

(7) **DownSa**(Z) shifts a transition having the parent state of Z as source state to Z. The parent state of Z must be an AND-state.

(8) **DownTa**(Z) shifts a transition having the parent state of Z as target state to Z. The parent state of Z must be an AND-state.

Both transformations simply replace the parent state of Z in the source (target) states of the transition by Z.

(9) **DownSas**(Z) transforms the transition to a synchronizing one and adds Z as further source states of the transition. The parent state of Z must be an AND-state and source state of the transition.

(10) **DownTas**(Z) transforms the transition to a synchronizing one and adds Z as further target states of the transition. The parent state of Z must be an AND-state and target state of the transition.

Example: We shifted the transition t5 to the AND-state book administration as described above. Now we would like to shift the transition back to the original source states. First we use t5.DownSas(book registration), which adds book registration as a new source state and transforms t5 to a synchronizing transition. With t5.DownSa(book preparation) the AND-state book administration in the source states is replaced by the state book preparation. Afterwards we may shift t5 down from the OR-states to the substates using the transformation $DownSg$.

4.3. Combining and splitting transitions

(11) **ComSe**(t_1, t_2, t) combines two transitions t_1 and t_2 triggered by the same event to a transition t if the preconditions of t_1 and t_2 are equivalent and both have the same source and target states.

(12) **ComTe**(t_1, t_2, t) combines two transitions t_1 and t_2 triggered by the same event to a transition t if the postconditions of t_1 and t_2 are equivalent and both have the same source and target states.

(13) **SplitSe**(t, P_1, P_2, t_1, t_2) splits the transition t into the transitions t_1 and t_2. The parameters P_1 and P_2 are preconditions, their disjunction must be equivalent to the precondition of t.

(14) **SplitTe**(t, P_1, P_2, t_1, t_2) splits the transition t into the transitions t_1 and t_2. The parameters P_1 and P_2 are postconditions, their disjunction must be equivalent to the postcondition of t.

These transformations are defined according to the relation Ξ (compare definition 3.7, p. 5).

4.4. Combining and splitting states

(15) **Combine**(Z_1, Z_2, Z) combines two *basic* states Z_1 and Z_2 resulting in a new basic state Z.

If the states Z_1 and Z_2 are not *root* states, they must belong to the same parent state. The condition of Z is the disjunction of the ranges of Z_1 and Z_2. In all transition having Z_1 or Z_2 as source or target state Z_1 or Z_2 are replaced by the state Z. If Z_1 or Z_2 are source states of a transition the guard is replaced by the precondition of the transition. The combination of states results in a new state with a "wider" range, nevertheless we want that the transitions can only be applied to object satisfying the original precondition of the transitions.

Example: We combine the states book borrowed and book in text book collection to a state book on loan using the transformation Combine (book borrowed, book in text book collection, book on loan). The result of this transformation is shown in figure 5.

(16) **Split**(Z, B_1, B_2, Z_1, Z_2) splits a *basic* state Z into two basic states Z_1 and Z_2.

The parameters B_1 and B_2 are conditions. They must be disjoint and their disjunction must be equivalent to the range of Z. If Z is part of a state aggregation B_1 and B_2 must not violate the orthogonality constraint. Otherwise the transformation is rejected. The condition of Z_1 equals to B_1, those of Z_2 to B_2.

If Z is not a *root* state Z must not be source or target state of transitions. Otherwise (Z is a root state) each transition having Z as source state is duplicated. In the source states of the original transition Z is replaced by Z_1 in the

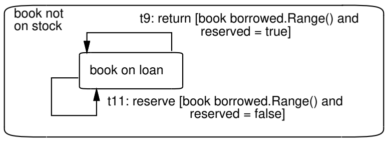

Figure 5. Splitting a state

duplicated one Z is replaced by Z_2. If Z is a target state the transition must be duplicated too and the target states are adopted. Furthermore the postcondition of the original transition is replaced by the conjunction of its postcondition with the range of its new target state (analogous to the duplicated transition).

The restriction that a state Z, which is part of a state hierarchie, must not be a source or target state of a transition is not very extensive. Suppose have combined two states as shown in the example of figure 5. As the new state is source and target state of transitions we are not able to split the state into the original states. However, we may use a combination of schema transformations to shift the transitions from the substates to the parent state and split the state afterwards.

4.5. Generating and Decomposing state generalizations

(17) **Geng**$(Z_1 \dots Z_i, G)$ produces a state generalization based upon the states $Z_1 \dots Z_i$ with the new OR-state G.

The states $Z_1 \dots Z_i$ must be *root* states or substates of the same OR-state. The transformation introduces a new OR-state G with the substates $Z_1 \dots Z_i$. Transitions remain unchanged by the schema transformation.

(18) **Decg**(G) decomposes a state generalization with the OR-state G. The OR-state must not be a substate of an AND-state. G must not be source or target state of transitions.

Example: In our example of figure 2 we can decompose the state generalization with the OR-state book not on stock. However, as this state is a source and target state of transitions we first have to shift down the transitions from the OR-state. Then we may decompose the state generalization simply by removing the OR-state with the transformation Decg(book not on stock). In a second step we may generalize the (now) root states book borrowed and book in text book collection using Geng(book borrowed, book in text book collection, G).

4.6. Generating and Decomposing state aggregations

(19) **Gena**(Z, n, A) builds a state aggregation with the AND-state A based upon the basic state Z .

Beyond the AND-state A, n OR-states $G_1 \dots G_n$ as substates of A are created. Each OR-state G_i has exactly one basic substate Z_i. The condition of a substate Z_i equals to the range of Z. According to the definition of the range of a state (compare definition 2.1) the ranges of A, G_i and Z_i are equivalent.

In all transitions having Z as source or target state Z is replaced by A. Afterwards Z is deleted.

(20) **Deca**(A, Z) decomposes a state aggregation with the AND-state A resulting in the basic state Z. All substates of A must have exactly one basic substate. None of the states of the state aggregation except the AND-state A must be source or target state of transitions.

The condition of Z equals to the range of A. In all transitions having A as source or target state A is replaced by Z. Afterwards A and all other substates of the state aggregation are deleted.

Example: Consider the AND-state book administration which should be decomposed. As there are states within this state aggregation which are source or target states of transitions we first shift them to the AND-state. E. g. the transition place is shifted to the AND-state as described in section 4.2. Afterwards the state aggregation is replaced by an equivalent basic state Z using the schema transformation Deca(book administration, Z). In a second step we would like to reintroduce the state aggregation again. We use Gena(Z, 2, book administration) resulting in a new state aggregation consisting of the AND-state and two OR-states as substates. Each OR-state has exaclty one basic substate (let's call them Z_1 and Z_2). These basic states could be split in two states. For instance we split Z_1 into two states using the transformation Split(Z1, book in catalogue.Range(), book registered.Range(), book in catalogue, book registered). Analogous we may split Z_2. Then the transitions can be shifted down from the AND-state to the original states.

4.7. Deleting and combining transitions

(21) **DelEx** deletes transitions whose pre- or postconditions result in $false$ (compare definition 3.7).

(22) **ComEx** combines transitions triggered by the same event having equal source states (target states) and equivalent postconditions (preconditions) by the disjunction of their pre (postconditions) (compare definition 3.7).

Based on this schema transformations we define the combined schema transformation $Clean$, which deletes and combines transitions of a statechart after a schema transformation has been applied.

4.8. Properties of the schema transformations

The main property of the presented schema transformation is that they are equivalence transformation (for the proofs we refer to [6]) which results in the theorem:

Theorem 4.1 If a correct statechart M_1 is transformed by applying one of the basic schema transformations into a statechart M_2 then $M_1 \equiv M_2$. □

For each schema transformation there exists an inverse schema transformation, which, however, may be a combination of transformations. For instance shifting a transitions from an OR-state produces several new transitions. For the inverse each of them must be shifted back to the OR-state and combined to one transition afterwards.

Finally, we can prove that the presented set of schema transformations is complete. This means that if two statecharts are equivalent, there is a sequence of basic schema transformations to transform one schema into the other. A consequence of this property is that the schema transformations we introduced suffice to derive any equivalent schema. This property is expressed in the following theorem:

Theorem 4.2 If two correct statecharts M_1 and M_2 are equivalent, than there exists a sequence of schema transformations to transform M_1 into M_2 and vice versa. □

5. Conclusion

We presented a formalization of a model for representing the dynamic behavior of objects. We present a metamodel and define the (abstract) semantics of statecharts as partial specification of methods. This allows the definition of the equivalence of statecharts. The main contribution of this work is the development of a complete set of basic schema transformation which maintain the semantics. The presented set of transformations suffices to derive any equivalent dynamic model from a given one.

There are several applications for the presented methodology. It serves as sound basis for design tools. It enables analysts and designers to start from an initial model and improve the quality of the model step by step. We can provide automatic support to achieve certain presentation characteristics of model. A model can be transformed to inspect it from different points of view. In particular a model suitable for conceptual comprehension can be transformed to a model better suited for implementation similar to the transformation of static conceptual models to logical models.

Our main application and motivation for the development of the model was to support automatic integration of partial models. This is the extension of the view integration approach in conceptual modeling to also incorporate dynamic models ([7, 8]).

References

[1] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. The Benjamin/Cummings Publishing Company, Inc, 1992.

[2] G. Booch. *Object-Oriented Design with Applications*. Benjamin Cummings, 1991.

[3] D. Coleman, P. Arnold, S. Bodoff, C.Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall Object-Oriented Series. Prentice-Hall, Inc, 1994.

[4] D. G. Firesmith. The inheritance of state models. *Report on Object Analysis and Design (ROAD)*, 2(6):13 – 15, Mar. 1996.

[5] A. Formica, H. Groger, and M. Missikoff. Object-oriented database schema analysis and inheritance processing: A graph-theoretic approach. *Data- and Knowledge Engineering*, 24:157–181, 1997.

[6] H. Frank and J. Eder. A meta-model for dynamic models. Technical report, Institut für Informatik, Universität Klagenfurt, Mar. 1997. http://www.ifi.uni-klu.ac.at/cgi-bin/show_an_abst?1997-05-FrEd.

[7] H. Frank and J. Eder. Integration of statecharts. In M. Halper, editor, *Third IFCIS International Conference on Cooperative Information Systems (CoopIS98)*, pages 364 – 372. IEEE Computer Society, Aug. 1998.

[8] H. Frank and J. Eder. Towards an automatic integration of statecharts. In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Metais, editors, *Conceptual Modeling - ER'99*, pages 430 – 444. Springer Verlag, Nov. 1999. (LNCS 1728).

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[10] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514 – 530, May 1988.

[11] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293 – 333, Oct. 1996.

[12] G. Kappel and M. Schrefl. Inheritance of object behaviour - consistent extension of object life cycles. In J. Eder and L. A. Kalinichenko, editors, *Proceedings of the Second International East/West Database Workshop*, pages 289 – 300. Springer, Sept. 1994.

[13] H. Lam and M. Missikoff. On semantic verification of object-oriented database schemas. In *Proceedings of Int. Workshop on New Generation Information Technology and Systems - NGITS*, pages 22 – 29, June 1993.

[14] M. Missikoff and M. Toaiti. Mosaico: an environment for specification and rapid prototyping of object-oriented database applications. EDBT Summer School on Object-Oriented Database Applications, Sept. 1993.

[15] Rational Software et.al. Unified modeling language (uml) version 1.1. http://www.rational.com/uml, Sept. 1997.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International, Inc, 1991.

[17] M. Schrefl and M. Stumptner. Behaviour consistent refinement of object life cycles. In D. W. Embley and R. C. Goldstein, editors, *Conceptual Modeling - ER'97: Proceedings of the 16th International Conference on Conceptual Modeling*, Lecture Notes in Computer Science 1313, pages 155 – 168. Springer, Nov. 1997.

[18] M. von der Beeck. A comparison of statecharts variants. In L. de Roever and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128 – 148, New York, 1994. Springer-Verlag.