Towards an Automatic Integration of Statecharts

Heinz Frank and Johann Eder

Universität Klagenfurt, Institut für Informatik-Systeme
Universitätsstraße 65-67, A-9020 Klagenfurt
{heinz, eder}@ifi.uni-klu.ac.at
http://www.ifi.uni-klu.ac.at

Abstract. The integration of statecharts is part of an integration methodology for object oriented views. Statecharts are the most important language for the representation of the behaviour of objects and are used in many object oriented modeling techniques, e.g. in UML ([23]).
In this paper we focus on the situation where the behaviour of an object type is represented in several statecharts, which have to be integrated into a single statechart. The presented approach allows an automatic integration process but gives the designer possibilities to make own decisions to guide the integration process and to achieve qualitative design goals.

1 Introduction

Conceptual modeling of a universe of discourse using an object oriented data model has two dimensions: the structure of objects and their relationships are represented in a static model (or object model) and the behaviour of objects is documented in a dynamic model ([3, 7, 24]).

Statecharts, introduced by David Harel ([15–17]), are a popular method for designing the behaviour of objects. This concept is used in various design methodologies, e. g. OMT ([24]), OOD ([3]) or UML ([4, 23]).

View integration is a commonly followed technique for developing a conceptual model. The universe of discourse is described from the viewpoint of different user groups or parts of the systems resulting in a set of external models. In a second step these models are integrated into a common conceptual schema.

Many integration methodologies consider only the structure of objects, their attributes and the relationships between them. A survey of such approaches can be found in [5]. Some approaches also deal with the integration of methods (e. g. [13], [27]), but little work as been done concerning the integration of behavioural models.

Preuner and Schrefl ([22]) discuss the integration of object life cycles. They use so called *Object/Behaviour Diagrams* to represent the behaviour of objects. In difference to our approach they assume that states and activities can be identified by their names. Common behaviour of objects, described in different views, can be identified by the names of the constructs. Conflicts, such as naming conflicts, must be solved before the integration process. In our approach states and transitions are identified by using logical conditions (see section 2). Of course we have to deal with naming conflicts, e. g. that different events have the same name. However, such conflicts do not influence the integration process. They can be solved even at the end of the integration process. Furthermore, the usage of logical conditions allows to automate the integration process.

In an earlier paper ([10]) we presented an overview of our whole integration methodology for object oriented views. For this method we assume that models have been developed from different perspectives. Each of these view models consists of a structural (or static) model and a set of behavioural (or dynamic) models in form of statecharts (for each type one). Our integration methodology consists of two major phases, the *integration of the static models* and the *integration of the dynamic models*.

The integration of the static models deals with the structural parts (types, attributes and their relationships to other types). The aim of this phase is to identify and solve conflicts (naming conflicts or structural conflicts) among the types of the various views. The result of this integration phase is the common conceptual static model of the universe of discourse. Several integration strategies, mainly for the entity relationship model ([6]), were published in the past, e. g. [1, 13, 14, 20, 26]. Further comparative analysis of view integration methodologies were made in [2] and [25]. For our methodology we have not yet developed another strategy for integrating the structural part of types but use already published methodologies, mainly that one of Navathe et al. ([20, 21]).

The integration of the static models results in an integrated conceptual model, a common agreement about types, their internal structure (attributes) and their relationships. Afterwards the integration of the dynamic models takes place. The input parameters are one integrated type and its various statecharts, describing the behaviour of this type from different viewpoints. The aim of this integration phase is to obtain the common behaviour of this type.

For the integration of statecharts we propose two phases:

- *The-integration-in-the-large:* In this integration phase an overall structure of the integrated statechart is developed. All statecharts of the integrated type are analyzed simultaneously in order to compute an integration plan. The integration plan consists of a tree of integration operators. Each operator has two statecharts as input and computes a statechart integrating both. Some of the integration operators can be performed automatically without the aid of a designer. In these cases the statecharts to integrate overlap only on marginal states (these are start and end states). The integration operators simply merge these marginal states to integrate the statecharts. The goal is to develop an integration plan with minimal integration effort, i. e. with minimal interactions with the designer.
  The integration-in-the-large phase was subject of an earlier paper ([12]), where we showed that this integration phase can be performed automatically without the aid of a designer.
- *The-integration-in-the-small:* According to the integration plan, the integration operators are carried out step by step. However, in some situations

we need a more detailed analysis of the involved statecharts and probably designer decisions.

The scope of this paper is the *integration-in-the-small.* We concentrate on the situation where we have to deal with one integrated type whose behaviour is described by two different statecharts. The result is the integrated statechart. In this paper however, we omit some formal details and proofs. Interested readers are refered to [9] and [11].

The paper is organized as follows: In section 2 we present an overview of the data model we use and discuss some extensions we made to the statechart language. In section 3 an example from the domain of a library is presented to demonstrate our methodology. The integration process for statecharts is shown in section 4, illustrated by the library example. In section 5 we draw some conclusions.

2 The data model

For the structural part of types we are using a very simple data model (according to [8]). 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 *integer* or using labels of user defined types.

For the representation of the behaviour of a type we are using the statechart language ([15–17]). A statechart of a type primarily consists of *states*, *events* and *transitions*. The major elements of statecharts are shown in figure 1.

A *state* is a condition or situation during 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 ([23]). We call this condition the *range* of a state.

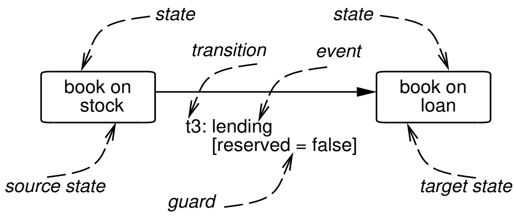

Fig. 1. The basic elements of statecharts

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 ([23]). The conjunction of the range of the source state of the transition and its guard is called the *precondition* of the transition. At the end of a transition the object satisfies the *postcondition* of the transition, which must imply the range of the target state of the transition.

As specification language for these conditions (the range of states, pre- and postconditions and guards of transitions) we use $\mathcal{TQL}++$ ([18],[19]). The 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.

As notation $S_1.Range()$ is used for the range of the state S_1. We use $t.PreC()$ for the precondition of a transition t. As these conditions are logical expressions we may combine them by disjunction, conjunction or negation. For instance, the precondition of a transition t is the conjunction of the range of its source state and its guard, that is $t.Source\_State.Range() \wedge t.Guard()$.

Ranges of states as well as pre- and postconditions of transitions are used to define the semantics of statecharts. We developed a complete set of schema transformations to transform a statechart into any other equivalent statechart ([11]). As an example we have transformations to decompose and to construct state hierarchies, to split and to combine states and to shift transitions within state hierarchies. These schema transformations are used in the integration process to prepare the statecharts and to integrate them.

In [11] we have shown that any statechart with state hierarchies can be transformed into an equivalent statechart without state hierarchies. For the integration we assume statecharts without state hierarchies.

Details about the semantics of statecharts and schema transformations can be found in [11] including the proofs that the schema transformations preserve the semantics of statecharts.

3 The example

Let us introduce a short example from the domain of a library showing the behaviour of books from the viewpoint of two departments. This example is used later to discuss each integration step.

Assume that the integration of the static models results into an integrated type Book, having the following syntax:

Book = [
    Title: str,
    Authors: {Author},
    reserved: bool,
    status: (new, in library, borrowed, in textbook collection)]

The behaviour of a book from the viewpoint of the Service Desk Department is shown in figure 2(a). The department registers new books and places them into the library. Books can be borrowed if they are available. Borrowed books are returned into the library.

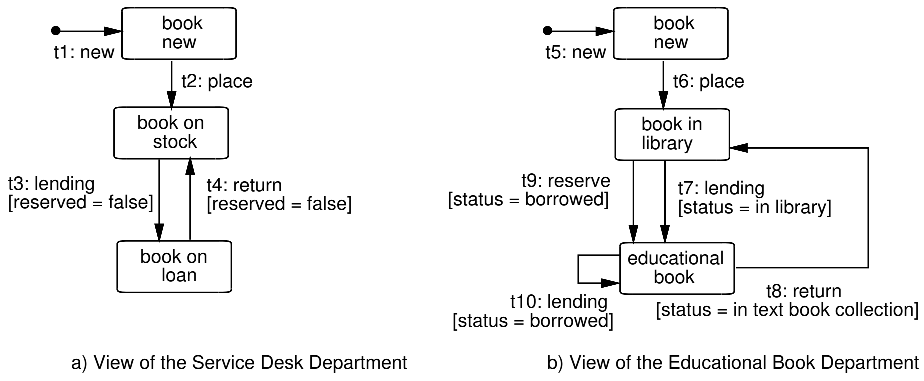

Fig. 2. The behaviour of the type book

Table 1. Ranges of the states of the Service Desk Department

book new	this.status = new
book on stock	this.status = in library $\wedge$ this.reserved = false
book on loan	this.status = borrowed

The second department is the Educational Book Department which registers new books and places them into the library too. Furthermore, it is responsible for the administration of educational books which are necessary for a lecture during a certain period. These books may not be borrowed by anyone until the end of the lecture. If a book in question is out of stock it can be reserved by the department. Such books may not be borrowed by anyone except the educational book department. If a book, borrowed by the department, is no longer necessary for a lecture it is returned. The behaviour of a book from the viewpoint of this department is shown in figure 2(b).

For the integration we require the specifications of the ranges of states which are shown in the tables 1 and 2. The postconditions of the transitions are either equivalent to the range of their target state or shown in table 3. Note that for the specifications of the guards we omit the keyword *this* in the figures.

4 The process of integrating statecharts

4.1 Overview

According to the integration plan, which was developed in the integration-in-the-large phase, we have to deal with one integrated type and two statecharts to integrate. Structural conflicts, such as naming conflicts, have already been solved in previous integration steps. Therefore, we do not have to consider naming conflicts between attributes used in the definitions of the ranges of states, pre- and postconditions or guards of transitions.

Table 2. Ranges of the states of the Educational Book Department

book new	this.status = new
book in library	(this.status = in library $\vee$ this.status = borrowed) $\wedge$ this.reserved = false
educational book	(this.status = borrowed $\wedge$ this.reserved = true) $\vee$ this.status = in text book collection

Table 3. The postconditions of the transitions

t3:lending	this.status = borrowed $\wedge$ this.reserved = false
t6:place	this.status = in library $\wedge$ this.reserved = false
t7:lending	this.status = in textbook collection
t8:return	this.status = in library $\wedge$ this.reserved = false
t9:reserve	this.status = borrowed $\wedge$ this.reserved = true
t10:lending	this.status = in textbook collection

The integration process resulting in the integrated statechart consists of three major phases (shown in figure 3):

- The *schema analysis* phase: In this phase the statecharts are analyzed to find common behaviour which is represented in a graph called the *state relationship graph.*
- The *schema transformation* phase: The statecharts are transformed so that common behaviour is represented by a single state in both statecharts.
- The *schema merging* phase: The transformed statecharts are merged to the integrated statechart.
- The *schema restructuring* phase: The integrated statechart is restructured to meet quality criteria and to make it more readable. The result of this step is the final statechart.

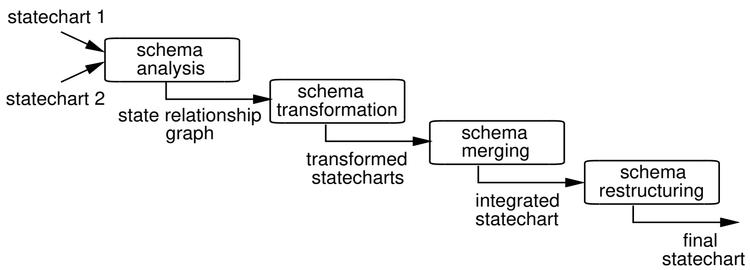

Fig. 3. The process of integrating two statecharts

4.2 The schema analysis phase

In the schema analysis phase the schemas are analyzed to find behaviour which is described in both schemas. The ranges of the states are used to detect common behaviour. We define four classes of relationships between states:

- Two states are *disjoint* if their ranges exclude each other. No object can satisfy the ranges of both states simultaneously.
- Two states are *equivalent* if their ranges are equivalent. Each object either satisfies both ranges or neither.
- A state S_1 *subsumes* a state S_2 if the range of S_2 implies the range of S_1. An object, satisfying the range of S_2, satisfies the range of S_1 too.
- Two states are *overlapping* if their ranges overlap. If an object satisfies the range of one of the states, it might but need not satisfy the range of the other state.

Example: The states book new from the book ordering department and book new from the book book service desk are equivalent as their ranges are equivalent ($this.status = new$). The range of state book on stock ($this.status = in\ library \wedge this.reserved = false$) implies the range of book in the library (($this.status = in\ library \vee this.status = borrowed) \wedge this.reserved = false$). The states book on loan ($this.status = borrowed$) and the state educational book (($this.status = borrowed \wedge this.reserved = true) \vee this.status = in\ text\ book\ collection$) overlap. (compare table 1 and 2). □

The relationships between states are used to build the *state relationship graph*. Nodes of the graph correspond to states of the statecharts, edges represent the relationship between two states and are annotated with the kind of relationship. An edge between two nodes exists if the corresponding states are not disjoint. Note that states of the same statechart must be disjoint if the statechart is correct ([17]). Figure 4 shows the state relationship graph of our library example.

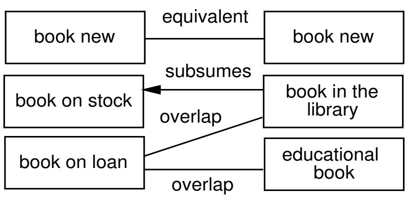

Fig. 4. The state relationship graph

4.3 The schema transformation phase

The goal of the schema transformation phase is to transform the statecharts such that common behaviour is described by a single state in both statecharts,

i. e. until the state relationship graph contains only 'equivalent' edges between nodes.

For transforming the statecharts we are using the schema transformation $Split(S, S_1, S_2, B_1, B_2)$ which replaces a state S with two states: S_1 with the ranges B_1 and S_2 with the range B_2, where $B_1 \vee B_2$ is equivalent to the range of S.

A transition t, having S as source state, is replaced with two transitions t_1 with source state S_1 and t_2 with source state S_2. Transitions, having S as target state, are replaced with two transitions in an analogous way. However, in order to preserve the semantics, the postcondition of t_1 is replaced with the conjunction of the postcondition of t and the range of its new target state S_1. Similar, the postcondition of t_2 is replaced with the conjunction of the postcondition of t and the range of S_2. Note that transitions having S as source and target state are replaced with four transitions.

After the transformation some of the copied transitions might have contradictory pre- or postconditions, e. g. if it is not possible for any object to satisfy the range of a source state and the guard of a copied transition simultaneously. Such transitions can be deleted, without changing the semantics of the statechart. More formal details about the schema transformation including the proof that *Split* preserves the semantics of a statechart can be found in [11].

This schema transformation and the information represented in the state relationship graph are used to transform the statecharts. We have to consider two cases: a state is subsumed by another state and states which overlap.

Transformation of subsuming states. A state S, subsuming a state T of the other statechart, is transformed into two states S_1 and S_2 such that the ranges of S_1 and T are equivalent and the ranges of S_2 and T are disjoint. We need the transformation

1. $Split(S, S_1, S_2, T.Range(), S.Range() \wedge \neg T.Range())$

Split demands that $S.Range() \equiv T.Range() \vee (S.Range() \wedge \neg T.Range())$. This is obvious, as the range of T implies the range of S.

In the state relationship graph the edge between S and T is deleted. The node S is replaced with the nodes S_1 and S_2. Obviously, S_1 has an equivalent relationship to T, but is disjoint to all other states. The node S_2 may have relationships to other states, if S had further (other than disjoint) relationships.

Example: According to the state relationship graph (figure 4) the state book in the library subsumes the state book on stock. The state is split into two states with $Split(book\,in\,the\,library, book\,in\,the\,library(1), book\,in\,the\,library(2), B_1, B_2)$. B_1 is the range of the state book on stock: *this.status = in library* $\wedge$ *this.reserved = false*. B_2 becomes *book in the library.Range()* $\wedge \neg$ *book on stock.Range()* which is equivalent to *this.status = borrowed* $\wedge$ *this.reserved = false*. It is easy to see that the disjunction of B_1 and B_2 is equivalent to the range of the state book in the library (see tables 1 and 2). The ranges of the new states:

book in the library (1)	this.status = in library $\wedge$ this.reserved = false
book in the library (2)	this.status = borrowed $\wedge$ this.reserved = false

The result of the transformation is shown in figure 5. Note that all transitions starting from or ending in the original state are duplicated, their source and target states are adopted. To make it more readable we numbered them: t6(1):place is the first copy of the transition t6:place, t6(2):place the second one.

Lets take a closer look to the guard of the transition t9(1):reserve, which is *this.status = borrowed*. However, the range of the source state of this transition is *this.status = in library* $\wedge$ *this.reserved = false*. As the precondition of a transition is the conjunction of the range of its source state and its guard, no object can satisfy the precondition of the transition t9(1):reserve. We delete this transition and in analogy the transition t7(2):lending.

We have to consider the changed postconditions of the new transitions, e. g. the transition t8(2):return. The transformation replaces the postcondition of transitions, having the split state as target state, with the conjunction of its original postcondition and the range of its new target state. The postcondition of the transition t8:return is *this.status = in library* $\wedge$ *this.reserved = false* (see table 3). Taking the range of the state book in the library(2) we see that no object is able to satisfy the conjunction of the postcondition of t8:return and the range of this state. The transition t8(2):return is deleted and in analogy the transition t6(2):place.

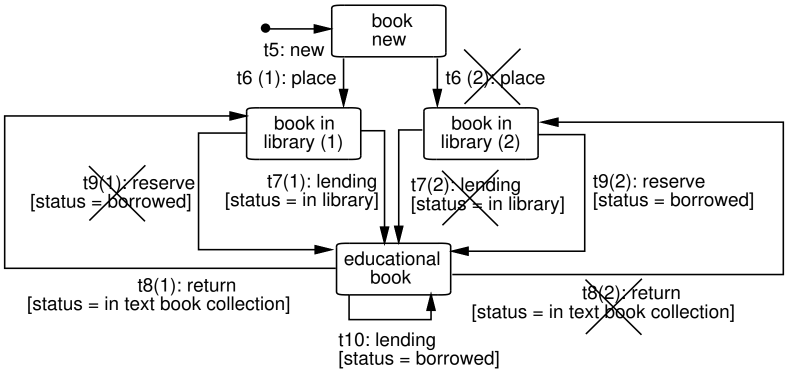

Fig. 5. Transforming the view of the Educational Book Department using the transformation *Split*

Finally, we have to change the state relationship graph shown in figure 6. Note that the relationship between book on loan and book in the library (2) is now 'subsuming' (compare table 1). □

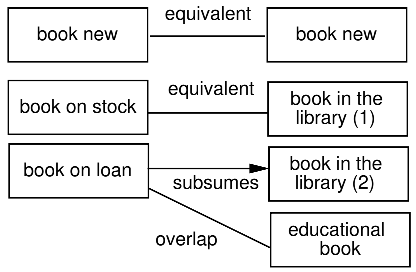

Fig. 6. The state relationship graph after transforming the view of the Educational Book Department

Transformation of overlapping states. In the case that two states S and T overlap, we split them into four states. Two states are disjoint to each other and two states are equivalent. The transformations:

1. $Split(S, S_1, S_2, S.Range() \wedge \neg T.Range(), S.Range() \wedge T.Range())$
2. $Split(T, T_1, T_2, T.Range() \wedge \neg S.Range(), T.Range() \wedge S.Range())$

The schema transformation *Split* demands that $S.Range() \equiv (S.Range() \wedge \neg T.Range()) \vee (S.Range() \wedge T.Range())$ which is obvious. It is easy to see that the analogous condition holds for the second transformation too.

In the state relationship graph S and T are replaced with nodes which correspond to the new states. Note that the states S_2 and T_2 are equivalent but disjoint to all other states. The states S_1 and T_1 might have relationships to other states (if S or T had further relationships other than disjoint ones to other states).

Example: In the state relationship graph shown in figure 6 we have an 'overlapping' relationship between the states book on loan and educational book. The states are split as shown in figure 7 and figure 8.

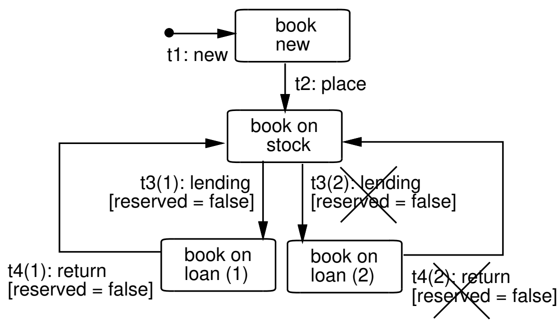

Fig. 7. Transforming the view of the Service Desk Department using the transformation *Split*

The state book on loan is split using the transformation *Split(book on loan, book on loan(1), book on loan(2),* B_1*,* B_2*)* where B_1 = *book on loan.Range()* $\wedge$ $\neg$ *educational book.Range()* and B_2 = *book on loan.Range()* $\wedge$ *educational book.Range()*. The new states with their ranges:

book on loan (1)	this.status = borrowed $\wedge$ this.reserved = false
book on loan (2)	this.status = borrowed $\wedge$ this.reserved = true

The transition t4(2):return is deleted (figure 7) because its guard does not match with the range of its source state. The transition t3(2):lending is deleted because its postcondition does not imply the range of its source state (compare table 3).

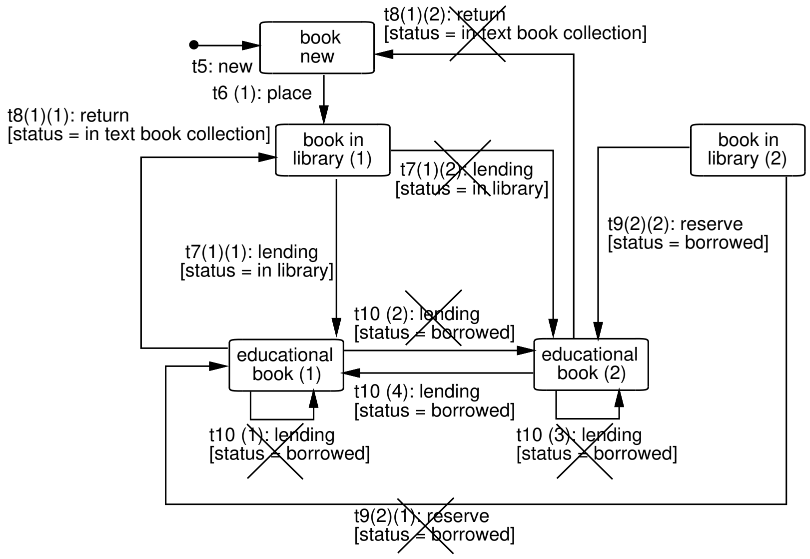

Fig. 8. The second transformation of the view of the Educational Book Department using the transformation *Split*

The state educational book is split using the transformation *Split(educational book, educational book(1), educational book(2),* B_1*,* B_2*)*. The parameter B_1 is *educational book.Range()* $\wedge$ $\neg$ *book on loan.Range()*. B_2 is *educational book.Range()* $\wedge$ *book on loan.Range()*. The new states with their ranges:

educational book (1)	this.status = in textbook collection
educational book (2)	this.status = borrowed $\wedge$ this.reserved = true

Note that the transition t10:lending is replaced with four transitions as the state educational book is source and target state of this transition (see figure 8).

The transitions t10(1):lending and t10(2):lending are deleted because the range of their source states and their guards contradict. The postcondition of the transition t9(2)(1):reserve (see table 3) does not imply the range of its target state and is removed too.

The transition t7(1)(2):return has a guard which does not match with the range of its target states. The transitions t10(3):lending and t7(1)(2):lending have postconditions which do not imply the range of their target states (compare table 3). All these transitions are deleted.

Finally, we change the state relationship graph again (figure 9). Note that the states book on loan(1) and book in the library(2) now have equivalent ranges ($this.status = borrowed \wedge this.reserved = false$). The schema transformation phase of our example is completed, the state relationship graph contains only nodes representing disjoint or equivalent states. □

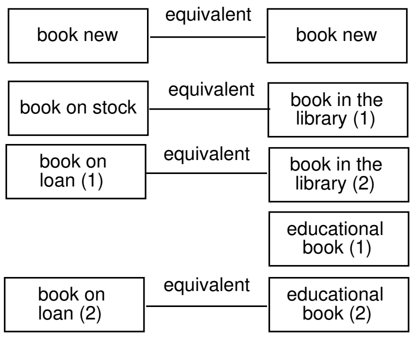

Fig. 9. The final state relationship graph

With the presented schema transformations the statecharts are transformed as long as the state relationship graph contains subsuming or overlapping edges between nodes. In each step common behaviour is extracted and a subsuming or overlapping edge of the relationship graph is removed. The transforming process terminates in any case.

4.4 The schema merging phase

In this phase the transformed statecharts are merged into the integrated statechart. According to the state relationship graph, states are either disjoint or equivalent. The merging of the statecharts is quite easy. Two equivalent states are merged to a single state, disjoint states remain unchanged.

Example: Let us complete the integration process of our library example. The statecharts (figure 7 and figure 8) are merged. Equivalent states such as book on stock and book in the library(1) (figure 9) are merged to single states. The integrated statechart is shown in figure 10. □

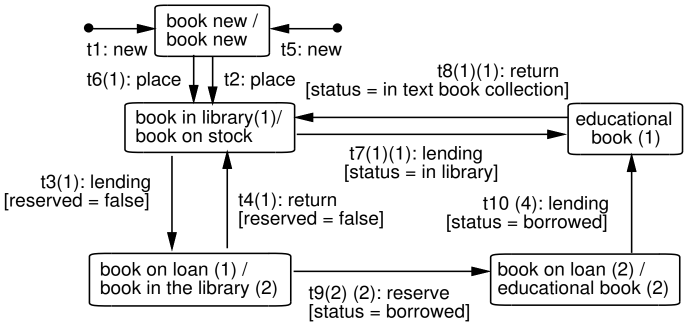

Fig. 10. The integrated statechart

During the integration process we used schema transformations that transform states into disjoint states. As shown in [11] a state of a statechart can always be split into two disjoint states without changing the semantics of the statechart. It is easy to see that combining equivalent states does not change the information described by a statechart. The information, described by the statecharts which must be integrated, is described by the integrated statechart too.

4.5 The schema restructuring phase

The schema merging step delivers a first cut integrated statechart. Since we made a series of transformations to extract common behaviour, the readability of the integrated statechart should be improved, e g. redundant information should be removed.

For instance, the transitions t1:new and t5:new have equivalent pre- and postconditions and the same target states. Assuming that they are triggered by the same event, i. e. new, we may combine them to one transition. Otherwise the designer should rename one event. Analogous the transitions t2:place and t6(1):place can be combined to a single transition.

Before starting with the integration we have decomposed state hierarchies. In the schema restructuring phase state hierarchies can be introduced again. For instance, if states have some transitions triggered by the same event with the same source (or target) state in common, they could be put into a state generalization.

We can restructure the statechart automatically using some quantitative design goals like minimizing the number of edges, or minimizing the number of necessary guards. The final restructuring and naming, however, should be made by a (human) designer to meet qualitative design goals like readability.

5 Conclusion

In this paper we presented an approach for integrating statecharts, which is part of an integration methodology for object oriented views. We assume that the integration of the structural parts of an object oriented data model (i. e. the integration of the types) has already be done.

We concentrate on the situation where the behaviour of one type is described by two different statecharts. We show, how common behaviour can be found and how the different statecharts are merged into an integrated statechart. The integration process follows a very simple strategy: whenever two states are not disjoint, they are transformed into disjoint states and the common ground of these states is put into a separate state.

It was our aim to relieve the designer by automating the integration as much as possible. The major steps of our integration methodology can be done without the aid of a designer. The schema analysis phase, the schema transformation phase, and the schema merging phase could be performed by a tool. The crucial part of such a tool is the language, which is used to specify the range of states and the pre- and postconditions and guards of transitions. Assuming that this language is decidable, the integrated statechart can be computed. However, the designer is needed to make changes in order to improve the quality in the schema restructuring phase, e. g. to make it more readable.

At the moment we are working on a language for representing conditions on states and transitions which is decidable. In a next step we are planning to implement our integration methodology.

We see the main advantages of our approach in the formal treatment of the integration process which allows an automatic integration while giving the designer the possibilities to make decisions and their consequences are carried out automatically in the model.

References

1. Batini, C., Lenzerini, M.: A Methodology for Data Schema Integration in the Entity Relationship Model. IEEE Transactions on Software Engineering, 10(6):650–664, November 1984.
2. Batini, C., Lenzerini, M., Navathe, S.: A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys, 18(4):323–364, December 1986.
3. Booch, G.: Object-Oriented Design with Applications. Benjamin Cummings, 1991.
4. Booch, G., Rumbaugh, J., Jackobson, I.: The Unified Modeling Language: User Guide. Addison-Wesley, 1999.
5. Bukhres, O., Elmagarmid, A. (eds.): Object-Oriented Multidatabase Systems: A Solution for Advanced Applications. Prentice Hall, 1996.
6. Chen, P.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transaction on Database Systems, 9–36, March 1976.
7. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P.: Object-Oriented Development: The Fusion Method. Prentice Hall Object-Oriented Series. Prentice-Hall, Inc, 1994.

8. Formica, A., Groger, H., Missikoff, M.: Object-Oriented Database Schema Analysis and Inheritance Processing: A Graph-Theoretic Approach. Data- and Knowledge Engineering, 24:157–181, 1997.
9. Frank, H.: View Integration für objektorientierte Datenbanken. Dissertationen zu Datenbanken und Informationssystemen Band 32. infix, 1997.
10. Frank, H., Eder, J.: Integration of Behaviour Models. Proceedings of ER'97 Workshop on Behavioral Modeling and Design Transformations. http://osm7.cs.byu.edu/ER97/workshop4, November 1997.
11. Frank, H., Eder, J.: Equivalence of Statecharts. Technical report, Institut für Informatik-Systeme, Universität Klagenfurt, 1998.
12. Frank, H., Eder, J.: Integration of Statecharts. In Halper, M. (ed.): Third IFCIS International Conference on Cooperative Information Systems (CoopIS'98). 364–372. IEEE Computer Society, August 1998.
13. Geller, J., Perl, Y., Neuhold, E., Sheth, A.: Structural Schema Integration with Full and Partial Correspondence Using the Dual Model. Information Systems, 17(6):443–464, 1992.
14. Gotthard, W., Lockemann, P., Neufeld, A.: System Guided View Integration for Object-Oriented Databases. IEEE Transaction on Knowledge and Data Engineering, 4(1):1–22, January 1992.
15. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8:231–274, 1987.
16. Harel, D.: On Visual Formalisms. Communications of the ACM, 31(5):514–530, May 1988.
17. Harel, D., Naamad, A.: The Statemate Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology, 5(4):293–333, October 1996.
18. Lam, H., Missikoff, M.: On Semantic Verification of Object-Oriented Database Schemas. Proceedings of Int. Workshop on New Generation Information Technology and Systems - NGITS, 22–29, June 1993.
19. Missikoff, M., Toaiti, M.: Mosaico: An Environment for Specification and Rapid Prototyping of Object-Oriented Database Applications. EDBT Summer School on Object-Oriented Database Applications, September 1993.
20. Navathe, S., Elmasri, R., Larson, J.: Integrating User Views in Database Design. IEEE Computers, 185–197, January 1986.
21. Navathe, S., Pernul, G.: Advances in Computers. Vol. 35, Chapter Conceptual and Logical Design of Relational Databases, 1–80. Academic Press, 1992.
22. Preuner, G., Schrefl, M.: Observation Consistent Integration of Views of Object Life-Cycles. In Embury, S., Fiddian, N., Gray, W., Jones, A. (eds.): 16th British National Conference on Databases. Lecture Notes in Computer Science, Vol. 1405, 1998.
23. Rational Software et.al.: Unified Modeling Language (UML) Version 1.1. http://www.rational.com/uml, September 1997.
24. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice Hall International, Inc, 1991.
25. Schrefl, M.: A Comparative Analysis of View Integration Methodologies. In Traunmüller, R., Wagner, R., Mayr, H. (eds.): Informationsbedarfsermittlung und -analyse für den Entwurf von Informationssystemen. 119–136, 1987. Fachtagung EMISA.
26. Sheth, A.: Issues in Schema Integration: Perspective of an Industrial Researcher. ARO-Workshop on Heterogeneous Databases, September 1991.
27. Thieme, C., Siebes, A. Guiding Schema Integration by Behavioural Information. Information Systems, 20(4):305–316, 1995.