Consistency in Model Integration

Kees van Hee [1], Natalia Sidorova [1], Lou Somers[1], and Marc Voorhoeve[1]

[1] Eindhoven University of Technology, Dept. Math and Comp. Science, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
{k.m.v.hee, n.sidorova, l.j.a.m.somers, m.voorhoeve}@tue.nl

Abstract. We present a UML-inspired approach to modeling and analysis of complex systems. Different stakeholders of a system may have different views, modeled with different techniques. It is essential that the various aspect models (use cases and life cycles) provide a complete and consistent description of the total system. Our approach based on the composition and decomposition of (colored) Petri nets allows the integration of aspect models. We illustrate our approach by a case study.

1 Introduction

The analysis and engineering of complex systems cannot be performed by a single person. So, several system architects are involved, modeling various subsystems. Also the system will have several stakeholders with different views, which also requires various models for being able to validate the proposals of the architects. Different aspect models require different modeling techniques. UML [3] offers a wide range of such techniques, most of them being diagram techniques. A UML description of a moderate-size system contains hundreds of diagrams of various kinds. Each diagram models one or more aspects of the considered system. By concentrating on a few aspects at a time, validation by stakeholders becomes possible.

As the project proceeds, the aspect models will be integrated, while adding detail and rigor. This may lead to the discovery of inconsistencies. Early detection of such inconsistencies will help to reduce development costs, so the software industry is hard-pressed for methods to determine and preserve the consistency between the various models. We believe that there is no “silver bullet” for achieving this. The “honest” way is by a single model that integrates all aspects modeled so far. From this integrated model, the aspect models are derived as projections. If and only if such an integrated model can be found, the models made so far are consistent.

In this paper, we indicate how integrated models can be derived from aspect models in early stages of the development process. The key ingredients are Petri nets with synchronization and projection operators. We start with various aspect models that are combined by synchronization, resulting in an integrated model. From it, scenario's can be derived by projection in order to allow validation. There exist many

tools that support such an approach, some of them, e.g. ExSpect [7] and CPN [9], allow to add pre- and postconditions, resulting in a functional prototype.

The synchronization operator is closely related to the call mechanism for methods; the model thus can be used to support the design and implementation phases. We illustrate our proposal with a case study of the well-known library system, which is just large enough to illustrate the key aspects of our method.

In section 2, we introduce WF nets, a subclass of Petri nets used for our models and our operators for composing and decomposing them. In section 3, we describe the modeling, verification, and validation process. In section 4, we illustrate our process with a library case study, and we also show how our models can be extended, adding more functionality. We conclude with a comparison with related work.

2 Petri net models, synchronization and projection

We assume the reader has some knowledge of "classical" place-transition nets (bipartite directed graphs), markings (distributions of tokens in places) and the interleaving firing rule. A transition may fire in marking M iff it can consume the necessary tokens from M; as a result of this firing, a new marking M' is reached, consisting of M with the consumed tokens removed and produced tokens added. A net defines a reachability relation between its markings: a marking M' is reachable from a marking M iff a finite sequence of firings exists starting in M and ending in M'.

Marked nets are too general for modeling. In [1], the class of ***WF*** (workflow) nets is defined, which can be compared to UML activity diagrams. A WF net possesses a unique source and a unique sink place. Every node of a WF net, seen as a directed graph, lies on a directed path from the source to the sink place. A WF net possesses an initial marking (one token in the source place) and a final marking (one token in the sink place). It is ***sound*** iff (1) from the initial marking it is possible for every transition t to reach a marking where t may fire and (2) the final marking is reachable from any marking M that is reachable from the initial marking. Sound WF nets are the very nets used for modeling use cases and object life cycles, c.f. [4].

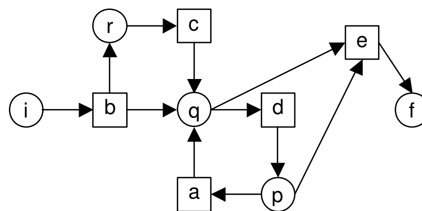

Fig. 1. Example of a WF net

In Fig. 1 a WF net is depicted. The places i,f are respectively the source and sink places. This net is sound, which can be verified by examining the reachable markings. For example, from the initial marking, a marking can be reached with only two

tokens in p (e.g. by firing b, then c then twice d). From this marking, it is possible to reach the final marking by firing a, then e.

For convenience, we omit in this paper the source and sink place from the WF nets. Thus a WF net has one or more start transitions (without input places) and end transitions (without output places). The firing of transitions models the occurrence of events. If a transition bordering the source place fires, a “create” event occurs, since in a sound net this transition does not consume any tokens. Analogously, destroy events are firings of transitions bordering the sink place. The soundness property now states that whatever is caused by a single creation can eventually be undone by a single destruction.

For Petri nets there are several methods for analyzing the behavior. Some of these methods use only the structure of the Petri net and not the underlying state space. T-invariants provide such a method. A T-invariant can be computed by standard linear algebra and can be related to sequences of transitions that return the system to the state before the sequence was executed. We use T-invariants in the validation process.

The tokens in the places refer to objects; every place contains references to objects of one and the same class. Initially, we abstract from the attributes of the objects, allowing “classical” analysis of our nets. Eventually, our models will consist of high-level nets, e.g. colored nets [9], specifying pre- and postconditions for the firing of transitions. A transition will fire only if its consumption satisfies the precondition; it will then produce tokens in accordance with the postcondition.

We add operators for composing and decomposing net models, which are essential for the integration of models and for checking their consistency. The composition operator is called ***synchronization*** and is indicated by a dotted line connecting two transitions. When transitions synchronize in a high-level net, data may be exchanged in either direction. In Fig. 2, an example net with synchronization is shown at the left. The synchronization result is the net in the middle, which is obtained by transition fusion: the transitions participating in a synchronization are glued together. This mechanism resembles the synchronization within process calculi like CCS [10]. Synchronization between sound nets does not always result in a sound net; the middle net in Fig. 2 is not sound, since transition cd cannot fire.

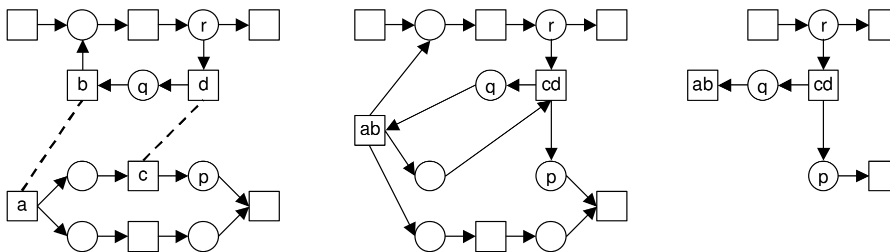

Fig. 2. Example of synchronization and projection

The decomposition operator is called ***projection***. Projection of a net w.r.t. a subset S of the net's places is obtained by removing all the places not in S plus the edges

leading to and from them. Transitions that become isolated are removed as well. The right-hand net shows the projection of the middle net w.r.t. the set $\{p,q,r\}$. If N is a connected net, P its set of places, and $S \subseteq P$, then N can be obtained by synchronizing its projections w.r.t. S and $P \setminus S$.

When creating WF nets for use cases, the transitions describe ***events*** that can occur. If a net models an object life cycle, the transitions represent ***methods*** of the object's class. In the design phase, the synchronization of methods will result in one of them calling the other. The following rule describes the essence of our approach: deriving an integrated model from aspect models and checking their consistency:

- The integrated model is derived from the aspect models (use cases and life cycles) by synchronization.
- All aspect models should be derivable from the integrated model by projection.

3 Modeling process

We focus on deriving an integrated logical model that captures the functionality of the system, we have left out other engineering activities. In general, we have a succession of ***elicitation, modeling, verification*** and ***validation*** steps. We split the modeling step into three steps: ***process*** modeling, ***data*** modeling, and ***transformation*** modeling. In the elicitation steps the stakeholders play an important role. There are several techniques to obtain useful information from a group of stakeholders. Well-known are “brown paper sessions” where stakeholders write down individually the most important items, like issues, functions, scenario's, or objects. These items are stuck to a brown paper board and grouped by the moderator into related groups. Then the items are discussed and terminology is fixed. These sessions are repeated with different topics. Group decision support systems [11] provide computerized support. The modeling step is done by system architects, who also perform verification, possibly “on the fly” during modeling using “correctness by construction”, sometimes after modeling (like verifying the integrated model). After modeling and verification comes validation with the help of stakeholders. As a result, a redesign may be needed.

The modeling, verification and validation steps are iterated until the stakeholders are satisfied with the logical model. At some stage when use cases have become stable, user interface designers can start to define screens containing forms and buttons. After having established the logical model, it is extended to accommodate for the designed user interface. We will describe the successive phases and steps in more detail. Remember that stakeholders are involved in phases 1 and 6 only.

Step 1: Elicitation

(a) Make a list of use cases, indicated by a name and some additional comments by the stakeholders.

(b) Define some allowed and explicitly forbidden scenario's (event sequences) for each use case.

(c) Identify the classes of objects that play a role in the scenario's.

(d) List relationships between object classes. The existence of these relationships is triggered by use case events that involve more than one object.
(e) Collect relevant attributes for the objects.
(f) Find static constraints that the system's state (the set of all living objects) should satisfy at all time.

Step 2: Process modeling
(a) Create WF nets for the use cases. Each WF net should combine the allowed scenarios for one use case and disallow the forbidden ones.
(b) Create WF nets for the object life cycles. The transitions are the methods of the classes.
(c) Integrate the workflows by identifying the transitions in use cases and object life cycles that must be synchronized. If necessary, adapt use cases and/or life cycles.

Step 3: Data modeling
(a) Construct the class model with relationships and attributes. We prefer functional relationships.
(b) Formalize the static constraints. Use logical predicates that can be translated back into natural language with increased precision. Add other common-sense static constraints.
(c) Define global variables. For each object class we define a global variable, called object store or object file. All objects that are active in the system reside in an object store. Also, other global variables like the current date or time are defined.

Step 4: Transformation modeling
(a) Combine the process model and the data model. Establish the relationships between object classes and methods. For each class we determine whether the methods create, read, update, or destroy objects from it (a CRUD-matrix).
(b) Determine the input and output parameters of the methods: places, global variables and additional parameters, e.g. for the user interface.
(c) Determine pre- and postconditions of the methods. The end product is the high-level integrated model.

Step 5: Verification
(a) Check the soundness of all workflows: use cases, object life cycles, and the integrated model.
(b) Check that all use case nets can be derived from the integrated model by projection.
(c) Check that each relationship in the class model is created somewhere.
(d) Check the preservation of the static constraints. Some constraints may be temporarily violated during the execution of a certain sequence of transitions (a transaction) but they should be valid after the transaction.
(e) If necessary, return to modeling.

Step 6: Validation
(a) Validate the integrated model by spawning new scenarios from T-invariants of the nets.

(b) Validate all static constraints.
(c) Present the scenario's with data transformations added.
(d) If necessary, return to one of the modeling steps.

Step 7: User interface integration
(a) Make additional classes and methods to accommodate the user interface.
(b) Synchronize the additional methods with the existing ones. If necessary, adapt the logical model.

The steps are not executed in the order presented; it is important that verifications and validations are effectuated as soon as possible in order to reduce costs. For example, step 5a should immediately succeed steps 2abc for the modeled WF nets. Usually, the nets created in 2ab can be verified by hand; the net in 2c often needs tool support [14]. Step 6a can succeed step 2c after verification. Indeed, we have drawn a rather sizable WF net depicting the described process. Afterwards, the logical model is translated into specifications for software components. These components can be constructed from scratch or the can be assembled from existing components. For component selection, the scenario's are helpful.

The steps above apply to systems of moderate size. Large systems should be split into subsystems to which the above steps apply. By synchronization the subsystems are integrated as suggested in section 5 of this paper. This extra integration step should be verified and validated similarly to the description above, concentrating on the interface between the subsystems. We will illustrate the above approach with an example case study.

4 Case study: a library system

In the case study we consider a more or less standard library system. Stakeholders are personnel and members that lend books. Several copies of the same book may exist. Members can make reservations for books that are not available. We focus on the modeling steps, in particular the process modeling step. Therefore we treat the other steps rather superficially.

4.1 Elicitation

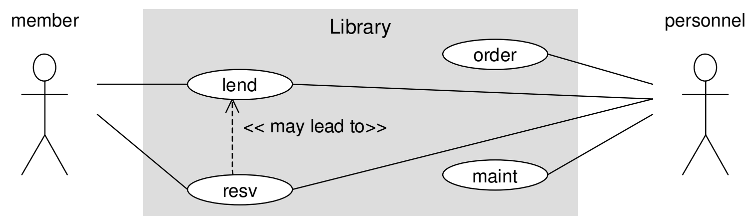

Fig. 3. Library use case diagram

Fig. 3 depicts typical use cases like lending a book, reserving and then lending a book, ordering books, and maintaining the member file and book catalogue. In Fig. 4 a loan/reserve use case net is given. The initial transition (event) is *s*, which creates a token in place *b* denoting the reservation by a member of a book in the catalogue. If the book is available, a loan is started (transition l_1). If the book is not available, the token stays in place *b* and if a matching book is returned, the reservation object can go to the notified state *d* by transition *n* (notification). From this state, transition l_2 can occur resulting in a loan (a token in *f*). A lent book can get lost (transition *lo*) or it will be returned (transition *re*).

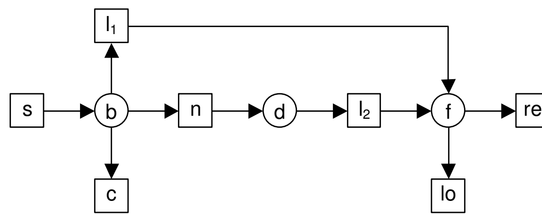

Fig. 4. Request / lend / reserve use case

Similar use cases can be found for maintenance and ordering activities. This is as complicated as it gets in our library case, but for other systems a use case may exhibit concurrent behavior, so it may have states that are distributed over various places. So far we encountered two object classes, reservations in places *b,d,* and loans in place *f*. When treating the other use cases, we encounter members, book orders, book copies, and book titles. It is necessary to distinguish book titles and copies, since several copies can exist for the same title. We determine the following classes (see Fig. 5):

MEM	library member
RSV	reservation **of** title **by** member
LOAN	loan **of** copy **by** member
TITLE	book title
BCPY	copy **of** title
ORD	order **of** title

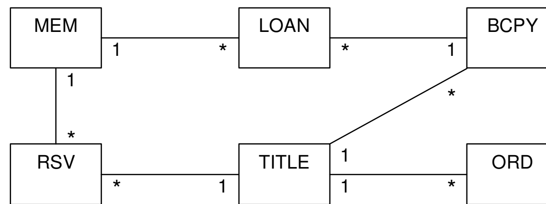

Fig. 5. Relations between object classes

4.2 Process modeling

The next phase is the modeling of each object's life cycle. A life cycle is composed of create, update and destroy methods, drawn as transitions. The objects correspond to tokens within places; an object class ranges over a set of places. The simple MEM objects only have one state *a*. Other objects may have more states, e.g. BCPY objects may be available for lending (*h*) or not (*g*).

Life cycle modeling starts with projecting the use cases onto the places from one class. The transition l_1 of our example use case thus becomes split into *ln* (creating a loan), *regl* (recording the loan of a book title) and *stal* (creating a loan object). By concentrating on one class, one is likely to find "gaps" in the life cycles found so far, which need to be plugged by adding transitions.

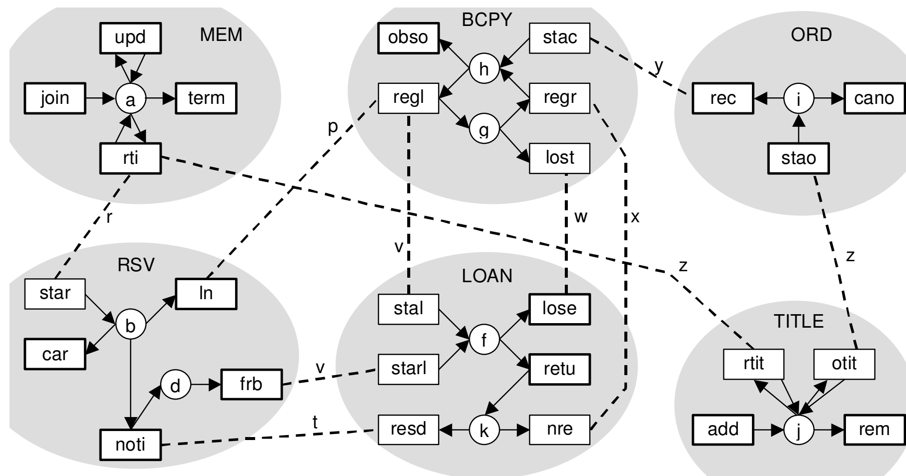

Fig. 6. Integrated library system model

The integrated model in Fig. 6 is obtained by synchronizing the transitions from life cycles that have been split (and some transitions that were added). Every life cycle produced so far should be obtainable by projection from the synchronization result. If not, the inconsistencies should be repaired (and discussed with the stakeholders).

The dashed lines indicating synchronizations are labeled; we will use these labels to identify the data exchanged in synchronization. Also, some transitions communicate with transitions from e.g. the user interface layer. These transitions have thick borders. The transitions, indicated by mnemonics, are explained in Table 1.

Table 1. Mnemonics for the transitions in the model

MEM	join	start membership
	upd	update member details
	rti	request title
	term	terminate membership
RSV	star	start reservation
	ln	immediate loan
	car	cancel reservation
	noti	notify member
	frb	fetch reserved book
LOAN	stal	start immediate loan
	starl	start reservation loan
	lose	lent book lost
	retu	book return
	nre	title not reserved
	resd	title reserved
BCPY	stac	start copy
	regr	register return
	lost	register loss
	regl	register loan
	obso	write off
ORD	stao	start order
	cano	cancel order
	rec	receive
TITLE	add	add title
	rtit	read title
	otit	order title
	rem	remove

In Fig. 7, the synchronizations have been spelled out for the submodel without the classes MEM, ORD and TITLE.

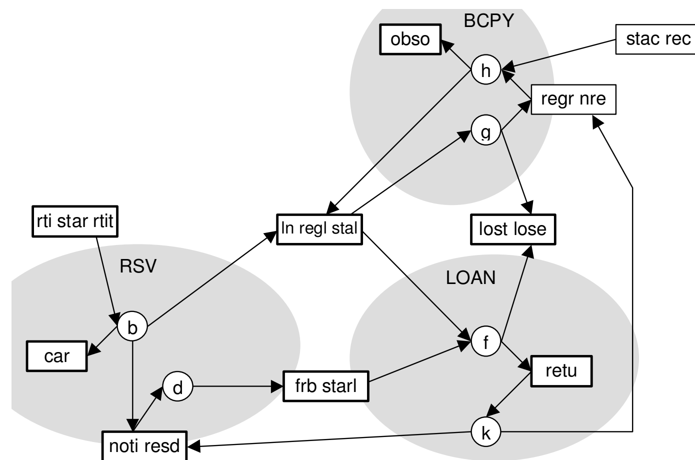

Fig. 7. Integrated model with spelled-out synchronizations (MEM, ORD, TITLE not shown)

4.3 Process verification and validation

A verification and validation step is possible before starting the data modeling. It is easy to see that all object life cycles are sound. For the use case in Fig. 4, this is also clear. Also, the use case in Fig. 4 can be obtained by projection on the places *b, d* and *f*.

We next turn to validation of the process model. It is possible to spawn "completed" scenarios by considering T-invariants of the net. A T-invariant is related to sequences of transitions that result in the same state before and after executing them. Each token produced is consumed and vice versa. T-invariant analysis is performed by standard linear algebraic techniques. Since completion of a T-invariant leaves no active token (case) in the net, all cases that were started have been completed, which makes T-invariants good candidates for validation. One rather intricate T-invariant is:

$$2(rti+req)+(ln+regl+stal)+(res+star)+retu+(resd+noti)+(frb+stal)+retu+(nre+regr).$$

This invariant indicates a scenario where two different members request the same book, one obtains a loan and the other a reservation. When the book is returned, the second member lends and finally returns it. The scenario is depicted as a sequence diagram in Fig. 8 and validated as such.

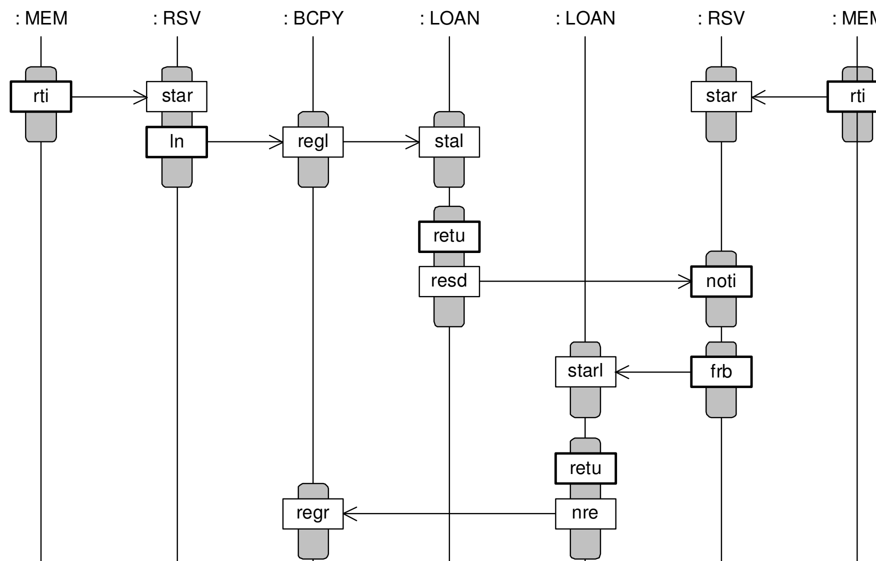

Fig. 8. Sequence diagram: loan after reservation

While validating the model in this way, omissions may be discovered. For instance, members may receive notifications for reserved books and fail to turn up to fetch them. After three days, the reservation expires and it is examined anew whether other reservations exist. Another omission is that after receiving ordered

copies, they should be examined for reservations just like returned lent copies. This leads to a redesign of the model: a loan terminates when the book is returned and the BCPY class is extended with states and transitions. This redesign is displayed in Fig. 9. The states of BCPY now become:

e	to be checked for reservations
k	awaiting notified member
h	free for lending
g	lent.

The transitions *resd* and *nre* move from class LOAN to BCPY and BCPY is extended with the transition *lres*, a reservation becoming a loan.

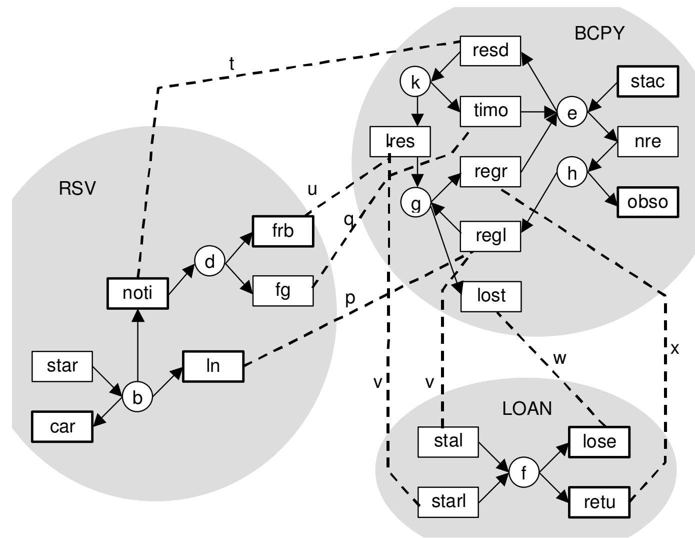

Fig. 9. Revised integrated model (MEM, ORD, TITLE not shown)

4.4 Data modeling

After constructing and validating the "classical" Petri net model, it becomes appropriate to consider data. The use case models and the events (transitions) that occur in them are helpful in eliciting the data involved. Data can be input data: attribute values of objects related to consumed tokens and input parameters (e.g. from the user interface). Output data are attribute values of objects related to production and output parameters.

By looking at the transitions connected to a certain class, we can produce a list of attributes for each class (Table 2). The boldface attributes are the object's key attributes; many-to-one relations are implemented by including the (foreign) key of the "one" object within the "many" object.

Table 2. Attributes of library classes

MEM	**lcode**: Tcode	membership nr
	name : Tname	name of member
	address : Taddr	address
RSV	**lcode**: Tcode	foreign key MEM
	ISBN: TISBN	foreign key TITLE
	date: Tdate	date of reservation
	state: $\{b,d\}$	
LOAN	**bcode**: Tcode	foreign key BCPY
	lcode: Tcode	foreign key MEM
	date: Tdate	loan date
BCPY	**bcode**: Tcode	key
	ISBN: TISBN	foreign key TITLE
	free: Boolean	available indicator
	indate: Tdate	date of acquisition
	state: $\{e,g,h,k\}$	
ORD	**ISBN**: TISBN	foreign key TITLE
	date: Tdate	order date
TITLE	**ISBN**: TISBN	key
	titdat: Ttitdat	author(s)/publisher/year/title

We can formulate constraints: for example, the "key constraint" that a member cannot have two reservations for the same title:

$$\forall\, r,r' : RSV \mid r \neq r' \bullet r.lcode \neq r'.lcode \vee r.ISBN \neq r'.ISBN \,.$$

Each synchronization in the integrated model will correspond to some method call where parameters and return values are exchanged. Therefore, for each synchronization these values must be specified:

p,v	lcode + bcode
q,t,y,z	ISBN
r,u	lcode + ISBN
w,x	bcode.

(Note that in deriving method calls from synchronizations a choice has to be made which object takes the initiative.) By looking at synchronizations that connect objects from different classes, we can verify the modeled relations. An object is often related to objects that are involved in its creation. For example, an ORD object is created from a TITLE object, which accounts for their relation in Fig. 5. Relations can be transferred when creating an object involves destroying another one. For example, a BCPY object is created from an ORD object and it "inherits" its relation to TITLE. The existence of a relation is often the condition for synchronization. The synchronizations t,q and u between RSV and BCPY transitions all have the condition that the book's title matches the reserved title.

4.5 Transformation modeling

The places in the WF nets contain tokens that correspond to objects. This makes it possible that one object corresponds to several tokens. When consumption and/or production of tokens occurs, the corresponding objects are created, destroyed, read or updated. When synchronization occurs, two or more objects can be accessed at the same time. In some cases, transitions (methods) must "globally" inspect all objects of a given class, which can be done by accessing a global variable containing all objects of this class. This variable has the same name as the class itself. It is not allowed to modify objects in this way; this should be done by the transitions.

For example, *resd* synchronizes with *noti* if a reservation exists and *nre* inspects the RSV variable to make sure that there are no reservations of the given title. If there are several reservations for the considered title, *noti* picks the oldest one, which also requires a global access. A third type of global access occurs when destroying MEM and TITLE objects, which may only occur if there are no other objects that refer to it. Another global variable is *day*, the current date. Table 3 lists which transitions create, read, update, and/or delete objects of each variable.

Table 3. CRUD matrix with transitions from the revised model

		MEM	RSV	BCPY	LOAN	ORD	TITLE	day
MEM	join	c						
	upd	u						
	rti	r					r	
	term	d	r		r			
RSV	star		c					r
	car		d					
	noti		u					r
	fg		d					r
	frb		d					
BCPY	stac			c				r
	resd			u				
	timo			u				
	nre		r	u				
	obso			d				
	regl			u				
	regr			u				
	lost			d				
	lres			u				
LOAN	stal				c			r
	star				c			r
	retu				d			
	lose				d			
ORD	stao					c		r
	cano					d		
	rec					d		
TITLE	add						c	
	otit						r	
	rtit						r	
	rem		r	r		r	d	

We are now in a position to specify every transition by giving pre- and postconditions. To this end we may use the Z language [8]. Each transition specification consists of a header and a body. The header describes the objects of the consumed and produced tokens, indicated by the place name decorated with a question mark (?) respectively exclamation mark (!) symbol. The synchronization labels are not decorated. Additional input and output parameters have been named *in* and *out* respectively. We do not mention global variables in the header.

Z requires that parameters are typed. Table 2 gives the types associated with the object attributes. In the body, conditions for the transition's occurrence are given. Unconnected conditions on different lines are interpreted as connected by conjunction ($\wedge$ symbol). Conditions only containing inputs must be preconditions; if they are not met, the transition will not occur. Many of the other conditions show how the output depends upon the input. Below we give the specifications for three transitions of the RSV class:

star

a reservation object is created from the synchronization *s*, with date and state added
$s : [l{:}Tcode, t{:}TISBN]; b! : RSV$
$(\nexists x : RSV \bullet x.lcode = s.lcode \wedge x.ISBN = s.ISBN)$ $b!.lcode = s.l \wedge b!.ISBN = s.t \wedge b!.date = day \wedge b!.state = b$

noti

the oldest RSV object matching synchronization *t* is selected and updated; the member and title id are output to the user interface
$t : TISBN; b?,d! : RSV; out : [l{:}Tcode, t{:}TISBN]$
$(\nexists r : RSV \bullet r.ISBN = b?.ISBN \wedge r.date < b?.date \wedge r.state = b)$ $t.ISBN = b?.ISBN \wedge d! = b? \oplus [state{:}d, date{:}day]$ $out = [l{:}b?.lcode, t{:}b?.ISBN]$

fg

an RSV object in state *d* waiting for more that 3 days is destroyed; its ISBN synchronizes via *q*
$q : TISBN; d? : RSV$
$d?.ISBN = q \wedge d?.date < day - 3$

Before validation, it can be e.g. verified that the reservation key constraint is preserved; when a new reservation id created by transition *star*, it is checked that no reservation with the same member and title code exists.

4.6 Extensions: user interface integration

While architects and stakeholders were busy with the logical model, another team of engineers has defined the user interface. It is now time to integrate the two models.

For instance, the “request title” (*rti*) transition requires the ISBN number of the requested title as input parameter. The user interface engineers have designed title selection screens to achieve this. These screens do not directly interface to transitions in the logical model, but to an additional class TQRY that allows a user to find out the ISBN number of a book he or she is interested in. The class TQRY has the following transitions (methods):

mqry	make query	the user describes his wishes
dres	display results	a list of titles matching the query is displayed
rqry	refine query	the original query is modified
selt	select title	a title is chosen from the list.

The *selt* transition will synchronize with the *rti* transition defined earlier. In Fig. 10, the life cycle of a TQRY object is given. As attributes, it has a predicate and a set of titles found so far that satisfy the predicate. The “ports” *r* and *z* of the *rti* transition have been connected to the *selt* transition and *dres* synchronizes with the TITLE class to filter out the titles matching the predicate. Synchronization thus allows to integrate the different models, also in the technical design phase.

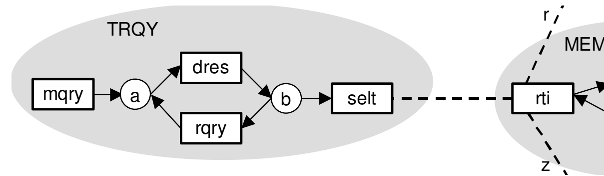

Fig. 10. The subnet for the request-title (*rti*) transition

5 Related work and conclusion

The use of Petri nets for the integration of UML models has been recommended by various authors. In all cases, some kind of composition operator is used to connect the various models. In [15], use case modeling with Petri nets is treated in conjunction with transition fusion (extended with place fusion). In [5], UML sequence diagrams that model scenarios are integrated within high-level Petri nets and used for prototyping. In [13], high-level nets are used for prototyping based upon state charts and collaboration diagrams. In [6], a thorough comparison of Petri nets and activity diagrams is given.

In our approach, the combination of synchronization and projection allows to move back and forth between aspect and integrated models, thus improving the consistency between the various aspect models. Current high-level Petri net tools like CPN [9] use token passing (i.e. place fusion) as composition operator. This operator adapts itself more easily to collaboration diagrams. The use of the synchronization operator (transition fusion) makes it easier to work with use cases, class and sequence diagrams. It also smoothens the transition to the design phases where method calls are implemented. Note that many modeling paradigms exist that allow syn-

chronization and projection within Petri nets, c.f. [2,12]. Any such paradigm will do for the purpose described here.

Acknowledgements

We are thankful for the support of our colleagues Ad Aerts, Tim Willemse and Jaap van der Woude and for the efforts of many of our students. It was the teaching of systems analysis and design throughout the years that led to the insights of the present paper.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In: P. Azéma and G. Balbo (eds) Proc. ATPN97, Lect. Notes in Comp. Science, Vol. 1248. Springer-Verlag, Berlin (1997)
2. E. Best, W. Fraczak, R.P. Hopkins, H. Klaudel, and E. Pelz. M-nets: an algebra of high-level Petri nets with an application to the semantics of concurrent programming languages. Acta Informatica Vol. 35, No. 10 (1998) 813-857
3. G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addsion-Wesley, Reading (1999)
4. M. Chaudron, K. van Hee, and L. Somers. Use Cases as Workflows. In: W. van der Aalst, A. ter Hofstede, and M. Weske (eds) Proc. BPM 2003, Lecture Notes in Comp. Science, Vol. 2678. Springer-Verlag, Berlin (2003) 88-103
5. Mohammed Elkoutbi and Rudolf K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In: M. Nielsen and D. Simpson (eds) Proc. ATPN 2000, Lect. Notes in Comp. Science, Vol. 1825. Springer-Verlag, Berlin (2000) 166-186
6. R. Eshuis and R. Wieringa. A Comparison of Petri Net and Activity Diagram Variants. In: Proc. Int. Coll. Petri Net Tech. Modeling Communication Based Systems (2001)
7. K.M. van Hee. Information Systems Engineering: a Formal Approach. Cambridge University Press, Cambridge (1994)
8. J. Jacky. The way of Z. Cambridge University Press, Cambridge (1997)
9. K. Jensen. Colured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. EATCS monographs on Theoretical Comp. Science. Springer-Verlag, Berlin (1992)
10. R. Milner. Communication and Concurrency. Prentice-Hall, London (1989)
11. J. Nunamaker, A. Dennis, J. Valacich, R. Vogel, and J. George. Electronic Meeting Systems to Support Group Work. CACM Vol. 34, No. 7 (1991) 40-61
12. L. Priese and H. Wimmel. A uniform approach to true-concurrency and interleaving semantics for Petri nets. Theoretical Comp. Science, Vol. 206, No. 1-2 (1998) 219-256
13. J. Saldhana and S.M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In: Proc. SEKE'00 (2000) 103-110
14. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. The Computer Journal, Vol. 44, No. 4 (2001) 246-279
15. J.L Woo, D.C. Sung, and R.K. Yong. Integration and Analysis of Use Cases using Modular Petri Nets in Requirements Engineering. IEEE Trans. on Software Engineering, Vol. 24, No. 12 (1998) 1115-1130