Available online at www.sciencedirect.com
SCIENCE DIRECT®
Data & Knowledge Engineering 56 (2006) 4–22
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Consistency in model integration

Kees van Hee *, Natalia Sidorova, Lou Somers, Marc Voorhoeve

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Received 9 February 2005; accepted 22 February 2005
Available online 26 March 2005

Abstract

State-of-the-art systems engineering uses many models reflecting various aspects of the modeled system. A major task of system engineers is to ensure consistency between the many models. We present an approach to the engineering of complex systems based on the modeling of use cases and object life cycles as Petri nets. Synchronization by place fusion allows the derivation of an integrated model that can be verified and validated. We illustrate our approach by a case study.

Keywords: Modeling; Petri nets; Synchronization; UML

1. Introduction

The analysis and engineering of a complex system usually requires the effort of several system architects, modeling various subsystems. The system will also have several stakeholders with different views, so various models are needed to validate the proposals of the architects. Often, these models address different aspects of the system, and require different modeling techniques. UML [3] offers a wide range of such techniques, most of them being diagram techniques. A UML

* Corresponding author.
E-mail addresses: k.m.v.hee@tue.nl (K. van Hee), n.sidorova@tue.nl (N. Sidorova), l.j.a.m.somers@tue.nl (L. Somers), m.voorhoeve@tue.nl (M. Voorhoeve).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2005.02.004

description of a moderate-size system often contains hundreds of diagrams of various kinds. By concentrating on a few models at a time, validation by stakeholders becomes possible.

As the project proceeds, the aspect models will be integrated, which 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 proper way is by providing a single model that integrates all aspects modeled so far. From this integrated model, the aspect models should be derivable 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. We represent various use cases and object life cycles as (simple) Petri nets and use synchronization (place fusion) to integrate them into a more complex net. The integrated model can be verified. Validation by stakeholders is possible by generating scenarios (connected event sequences). By using high-level Petri net tools, like ExSpect [7] or CPN [9], this approach can even be followed to derive a functional prototype of the system.

The synchronization operator can be implemented by the call mechanism for methods, so it will be possible 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, the subclass of Petri nets that we will use for our models. Also we define our operators for composing and decomposing WF nets. In Section 3, we indicate the steps taken for modeling, verification, and validation and the order in which to take them, i.e. the (user and system) requirement engineering process. In Section 4, we illustrate our approach with a library case study. We conclude with a comparison with related work.

2. Petri net models, synchronization and projection

We recall some basic facts about place-transition nets. A place-transition net is a bipartite directed graph. Its nodes are called *places* and *transitions*, depicted by circles and squares, respectively. A place p is an *input* place of transition t if there is an arrow leading from p to t. It is an *output* place if the arrow points from t to p. A state (also called marking) assigns a number of *tokens* to each place. A transition t is *enabled* if every input place of t is marked by one or more tokens. An enabled transition can *fire*, that is, it takes one token from each of its input places and adds a token to each of its output places. By firing a transition t in marking M, the marking becomes now M'. 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'.

We use a subclass called ***WF*** (workflow) nets (cf. [1]) for our models. WF nets can be compared to UML activity diagrams. A WF net possesses a unique source and a unique 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 called ***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. WF nets can be used to model use cases and object life cycles; these models should all be sound (cf. [4]). When creating WF nets for use cases, the

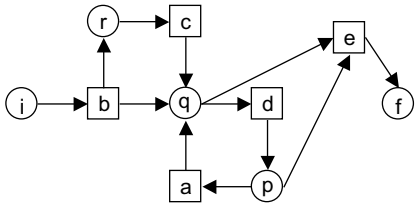

Fig. 1. Example of a WF net.

transitions describe ***events*** that can occur. If a net models an object life cycle, the transitions represent ***methods*** of the object's class.

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 $[i]$, the marking $[r, q]$ is reached by firing transition b. Next, by firing transition c, then twice d, the marking $[p^2]$ with only two tokens in p can be reached. 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 corresponds to the occurrence of events. The firing of a start (end) transition corresponds to an initial (final) create (destroy) event. We assume that initial events occur at and only at the start of an event sequence. Terminating event sequences are those sequences that end with a final event. If a WF net is sound, any possible event sequence can be continued to constitute a terminating sequence.

The simplest class of WF nets is the class of *state machine* WF nets. The start (end) transitions of such nets have one output (input) place and the other transitions have one input and one output place. State machine WF nets are sound, which is easy to prove since the reachable states correspond to singleton markings.

Petri nets can be analyzed in various ways. *Structural* properties refer to the structure of the Petri net, whereas *behavioral* methods use the state space (possible markings) and firings connecting the states. Generally, structural analysis methods are much more efficient than behavioral methods, due to the so-called state explosion that may occur. Soundness of a WF net is a behavioral property; despite the possibility of a state explosion, analyzing it is feasible for many nets occurring in practice (cf. [14]). Also, restricting oneself to a small set of construction patterns will lead to WF nets that are sound by construction [1]. In addition to the checking for soundness, we use the structural T-invariant analysis. The T-invariants can be computed by standard linear algebra techniques; they relate to sequences of transitions that lead from a certain state to itself. For example, $a + d$ constitutes a T-invariant in Fig. 1.

The main reason for using Petri net models is the existence of composition and decomposition operators. Transitions may possess (synchronous) ***ports*** and ***synchronization*** will occur for transitions whose ports are connected. When transitions in a net are synchronized, they must fire concurrently. Synchronizing transitions in two WF nets results in a WF net that can be obtained by transition fusion, as shown in Fig. 2. The two nets (with ports indicated) are shown at the left and the synchronization result is the net in the middle, which is obtained by fusing the transitions participating in synchronization. Usually we do not depict ports; rather we assume that they must

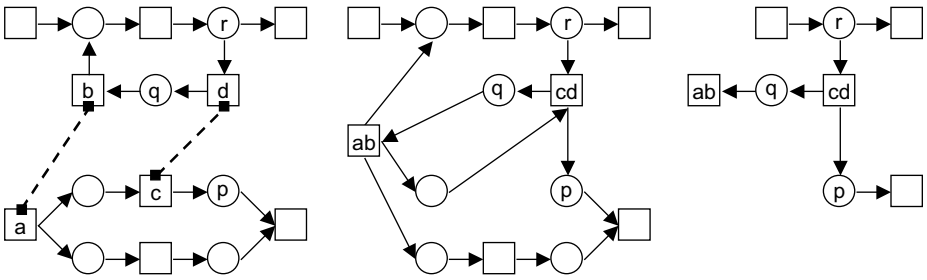

Fig. 2. Example of synchronization and projection.

exist when synchronization occurs. The synchronization operator in nets closely resembles the synchronization operator 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 (despite the apparent soundness of the original nets), since transition *cd* cannot fire. Any WF net can be constructed from state machine WF nets by the appropriate synchronizations. Models containing synchronizing state machine WF nets are equivalent to UML state chart diagrams.

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 is equal to the synchronizing related transitions from the respective projections of N w.r.t. S and $P \backslash S$.

Modeling by "classical" place-transition nets is appropriate in the early modeling stages, since it abstracts from details about object attributes. As the project proceeds, these details should be gradually added to the models. This is possible by using high-level nets (cf. [7,9]). Tokens modeling objects obtain ***attributes*** with values attached to them. Also, transitions possess typed ports that are used to exchange values with synchronizing transitions or with e.g. the user interface. Transitions obtain ***firing relations***, allowing to formulate pre- and postconditions for firing. This firing relation involves the values of tokens (objects) consumed and/or produced and of values received or sent through its ports. Synchronization occurs only if the received value at the one port equals the sent value at the other port connected to it. The implementation of synchronization by the call mechanism of programming thus becomes possible.

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 and 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, scenarios, or objects. These items are displayed and grouped into related groups by the moderator. 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, using patterns that are correct by construction wherever possible. Verification is performed where needed e.g. verifying soundness of 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 their activities. 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.

Step 1: Elicitation

(a) Make a list of use cases, each with its identification and a description by the stakeholders.
(b) Define some allowed and explicitly forbidden scenarios (event sequences) for each use case.
(c) Identify the classes of objects that play a role in the scenarios.
(d) List relationships between object classes. These relationships are often connected to use case events that involve more than one object.
(e) Collect relevant attributes for the objects.
(f) Find static constraints that a system's state (the set of all current objects) should satisfy at any point in time.

Step 2: Process modeling

(a) Create WF nets for the use cases. Each WF net should combine the allowed scenarios for a 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 synchronizing ports within the transitions of use cases and object life cycles. 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 as logical predicates that should hold in any state. Translating them in natural language allows validation by stakeholders.
(c) Define global variables. Each class possesses a global variable called object store, containing all current objects of that class. Other global variables like the current date or time may be needed.

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 ports.
(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 the proper projection.
(c) Check that object pairs are added to each relationship in the class model.
(d) Check the preservation of the static constraints. Some constraints may be temporarily violated, but should become valid again after the execution of a certain sequence of transitions (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 (and checking them).
(b) Validate all static constraints.
(c) Present the scenarios with data transformations added (by projection from the high-level integrated model).
(d) If necessary, return to one of the modeling steps.

Step 7: User interface integration

(a) Define additional classes and methods to accommodate the user interface. Typical classes contain session or dialogue objects that reflect the state of a user dialogue.
(b) Synchronize the user interface with the functional model at the user interface ports. If necessary, adapt either model.

The above steps need not be executed strictly, nor in the order presented. For instance, it is important to verify and validate as soon as possible in order to reduce costs. For example, the soundness check (step 5a) should immediately succeed WF net modeling (steps 2a,b,c), unless sound-by-construction nets like state machine WF nets were used. Checking the integrated net obtained in step 2c will often need tool support [14]. Step 6a can succeed step 2c after verification. The partial ordering of the steps can in fact be represented by a WF net.

After the above steps, the logical model is translated into specifications for software components. These components may need to be built, or may be assembled from preexisting components. For very large systems, a top-down modular approach is advisable. By decomposing the system into smaller subsystems, applying the above steps to each subsystem and synchronizing

them afterwards, we obtain an integrated model of the full system, that should be verified and validated as described above, concentrating on the interface between the subsystems.

4. Case study: A library system

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 reserve 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 depicts typical use cases like lending a book (possibly after reserving it), ordering a book, and maintaining the member file and book catalogue. In Fig. 4, the loan-and-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 a copy of that book is available, a loan is started (transition *ld*). If no copy is 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 *lr* 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*).

Similar use cases can be found for maintenance and ordering activities. We aim for so-called *state machine* nets for modeling use cases. By synchronizing two or more state machines, we

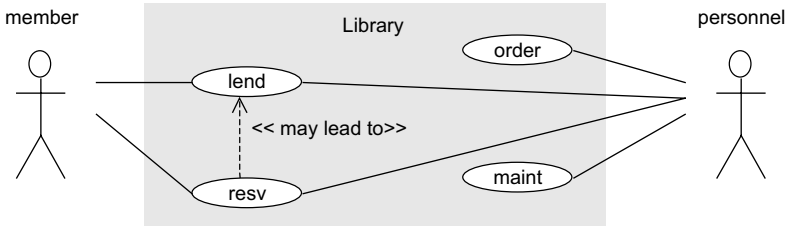

Fig. 3. Library use case diagram.

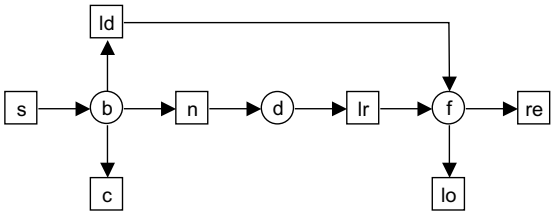

Fig. 4. Request/lend/reserve use case.

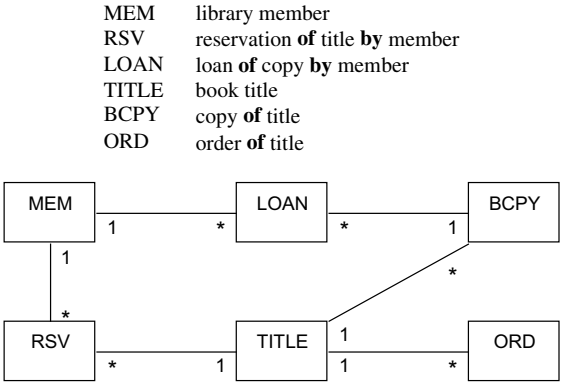

Fig. 5. Relations between object classes.

obtain nets that model concurrent behavior. So far we encountered objects of two 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. The classes and relations we determine are shown in Fig. 5.

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 a single class. The transition *ld* 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 filled by adding transitions.

The integrated model in Fig. 6 is obtained by synchronizing the transitions from life cycles that have been split (and possibly 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, which are labeled. We assume the existence at both ends of ports with the same label, which will identify the data exchanged during synchronization. Also, some transitions communicate with transitions from e.g. the user interface layer. These transitions have thick borders, indicating that they have ports that will be connected to the user interface. The transitions, indicated by mnemonics, are explained in Table 1.

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

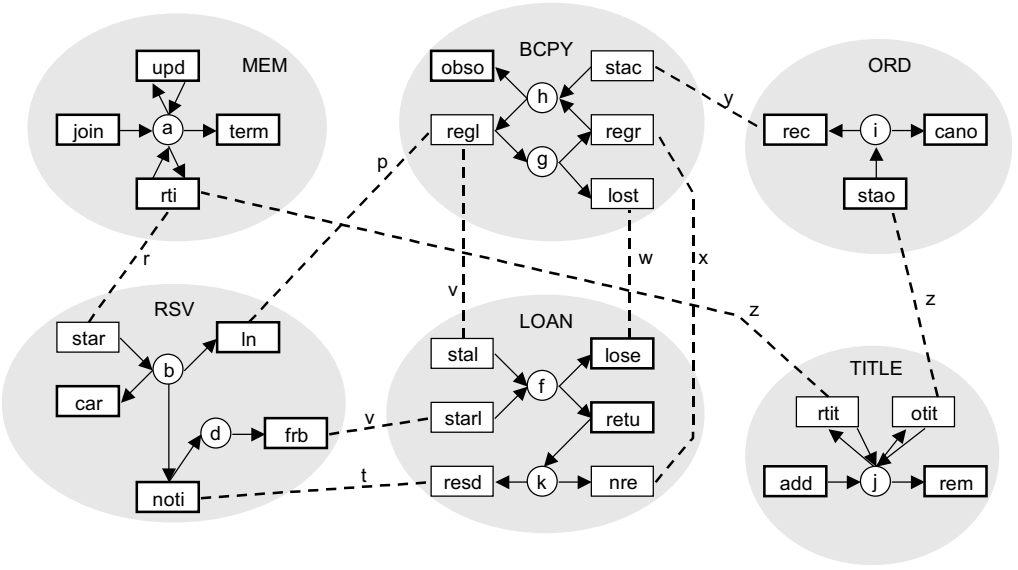

Fig. 6. Integrated library system model.

4.3. Process verification and validation

A verification and validation step is possible before the data modeling. The object life cycles and use case models are sound, being state machine WF nets. Also, the use cases can be obtained by projection. For instance projecting the net in Fig. 7 on the places *b*, *d* and *f* will result in the use case net in Fig. 4.

Next, we 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, the execution of which produces and consumes the same tokens. 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:

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

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.

While validating the model in this way, omissions may be discovered. It turns out that members, after having received a notification, may fail to turn up and claim the reserved book. After three days, the reservation expires and it is examined anew whether other reservations exist. Another omission is that after receiving an ordered copy from the bookstore, it should be examined for reservations just like returned lent copies.

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

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.

The loan-after-reservation sequence T-invariant now becomes

$$2(rti + star) + (\ln + regl + stal) + (retu + regr) + (resd + noti) + (frb + lres + starl) + (retu + regr) + nre,$$

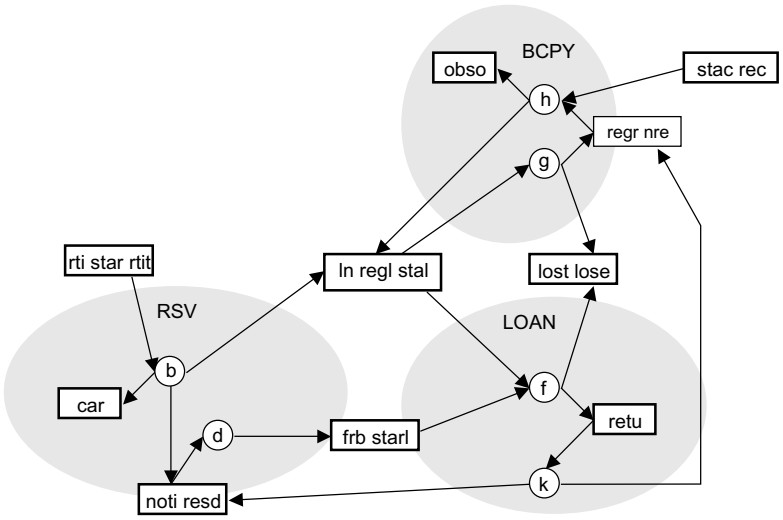

Fig. 7. Integrated model with spelled-out synchronizations (MEM, ORD, TITLE not shown).

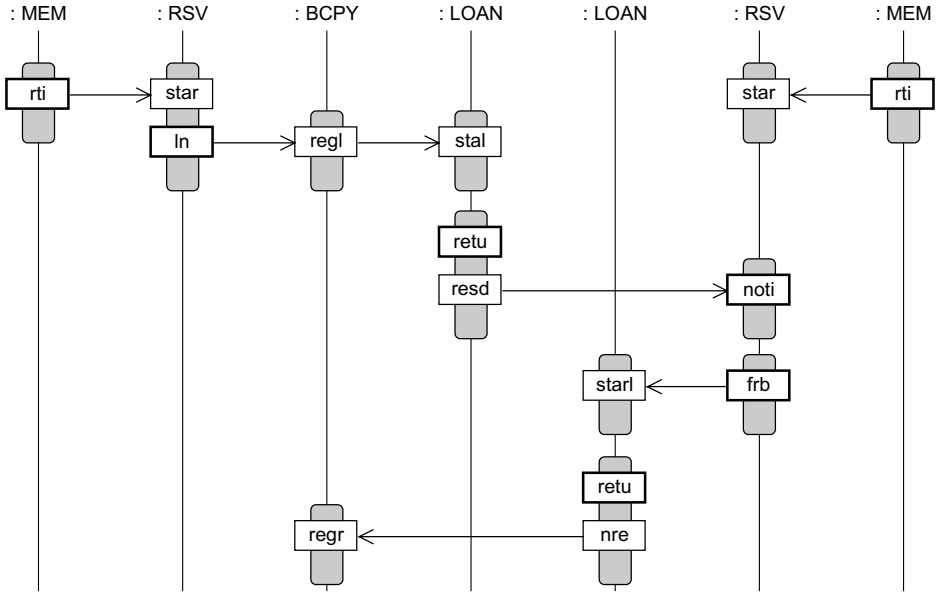

Fig. 8. Sequence diagram: loan after reservation.

which translates to the sequence diagram in Fig. 10. New T-invariants arise, like

$$2(rti + star) + (\ln + regl + stal) + (retu + regr) + (resd + noti) + (fg + timo) + (nre + regr),$$

which indicates the scenario where a reservation notification times out. These new scenarios deserve careful validation.

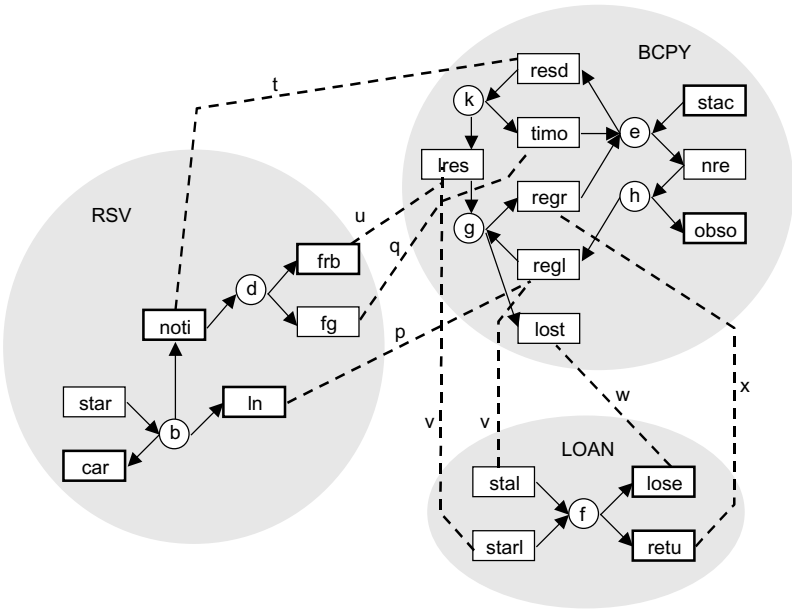

Fig. 9. Revised integrated model (MEM, ORD, TITLE not shown).

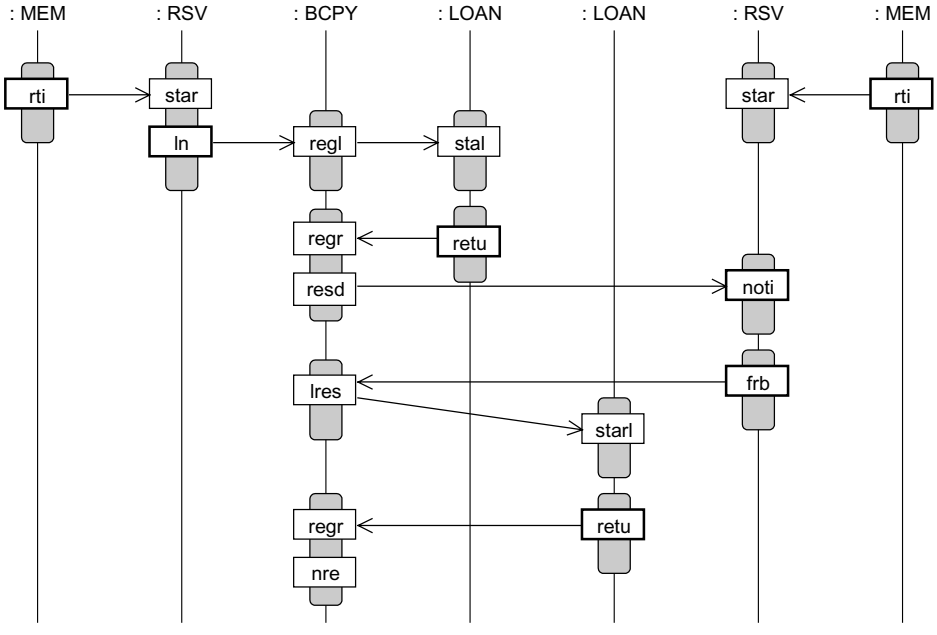

Fig. 10. Revised sequence diagram.

4.4. Data modeling

After constructing and validating the "classical" Petri net model, data will be added. The use case models and the events (transitions) that occur in them do help to elicit the data involved. Input data of a transition consists of attribute values of objects related to consumed tokens and input parameters (from the user interface or from synchronizations). Output data are attribute values of objects related to production and output parameters.

By considering the synchronizations between transitions connected to a certain class and use case transitions, we produce a list of attributes for each class (Table 2). Each object should contain key attributes (indicated in boldface); many-to-one relations are implemented by including the (foreign) key of the "one" object within the "many" object. We forgo a more sophisticated OCL-like notation that could have been used equally well.

We can formulate constraints: for example, the "key constraint" that a member cannot have two reservations for the same title. We use the Z notation (cf. [8]) to formulate this constraint, but any predicate language will serve as well.

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

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

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

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. If object life cycles have been modeled as state machine WF nets, there is a 1–1 correspondence between objects and tokens. When consumption and/or production of tokens occurs, the corresponding objects are created, destroyed, read or updated. When synchronization between methods occurs, two or more objects are simultaneously accessed. Occasionally, a method (transition) needs to inspect all current objects of a given class. A global read-only variable with the same name as the class is assumed to contain this set of current objects.

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 (i.e. book copies resp. reservations and loans) 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.

We are now in a position to specify the methods (transition) by giving pre- and postconditions. To this end, we use the Z language [8]. Each transition specification consists of a header and a body. The header contains a short description, followed by an indication of the consumed and produced token objects (indicated by the place name decorated with a question mark (?) respectively, exclamation mark (!) symbol), the parameters stemming from the synchronization ports (not decorated) and the user interface input and output ports (named *in* and *out*, respectively).

Z requires that parameters be typed. Table 2 gives the types associated with the object attributes. The body contains a list of conditions, which determine the firing relation. The transition may fire iff all conditions in its body hold. Often, the conditions are divided into preconditions and postconditions. Preconditions contain only input variables and state when a firing can occur. Postconditions state how the values of the output parameters are determined by the input

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*	

parameters. Below we give the specifications for three transitions of the RSV class. The *star* transition for example has the precondition $\nexists x : RSV \bullet x.lcode = s.lcode \wedge x.ISBN = s.ISBN$ and postconditions fixing the attributes of the new token $b!$.

star

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

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

It can be verified that e.g. the reservation key constraint is preserved: when a new reservation id is created by transition *star*, the precondition ensures that no reservation with the same member and title code exists.

4.6. 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 and dialogues to achieve this. These screens deal with a user interface object of 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	Renew query	A new (or refinement of the original) query is given
selt	Select title	A title is chosen from the list

The *selt* transition synchronizes with the *rti* transition defined earlier. In Fig. 11, 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 are connected to the *selt* transition and *dres* (and maybe also the query transitions) access the global TITLE class variable to select and/or display titles. Synchronization thus allows integration of the model to the user interface components selected and installed in the technical design phase. In the implementation, the *selt* method of class TRQY will call method *rti* of class MEM with the *lcode* of the member and the ISBN of the selected title.

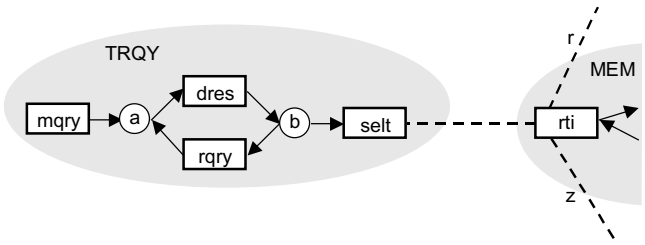

Fig. 11. The subnet for the request-title (*rti*) transition.

This example is typical for the integration of the user interface to the functional model. Similarly, other systems or components can be integrated, like EDI method calls, applets or servlets.

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. Token passing adapts itself more easily to collaboration diagrams while the synchronization operator (transition fusion) makes it easier to work with use cases, class and sequence diagrams. Synchronization also smoothens the transition to the design phases where method calls are used. Many modeling paradigms (cf. [2,12]) allow synchronization and projection within Petri nets. Any such paradigm will do for the purpose described here.

Acknowledgments

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, 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, E. Pelz, M-nets: An algebra of high-level Petri nets with an application to the semantics of concurrent programming languages, Acta Inform. 35 (10) (1998) 813–857.
[3] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addsion-Wesley, Reading, 1999.
[4] M. Chaudron, K. van Hee, L. Somers, Use cases as workflows, in: W. van der Aalst, A. ter Hofstede, M. Weske (Eds.), Proc. BPM 2003, Lecture Notes in Comp. Science, vol. 2678, Springer-Verlag, Berlin, 2003, pp. 88–103.
[5] M. Elkoutbi, R.K. Keller, User interface prototyping based on UML scenarios and high-level Petri nets, in: M Nielsen, D Simpson (Eds.), Proc. ATPN 2000, Lect. Notes in Comp. Science, vol. 1825, Springer-Verlag, Berlin, 2000, pp. 166–186.
[6] R. Eshuis, 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, Coloured 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, J. George, Electronic meeting systems to support group work, CACM 34 (7) (1991) 40–61.
[12] L. Priese, H. Wimmel, A uniform approach to true-concurrency and interleaving semantics for Petri nets, Theor. Comput. Sci. 206 (1-2) (1998) 219–256.
[13] J. Saldhana, S.M. Shatz, UML diagrams to object Petri net models: An approach for modeling and analysis, in: Proc. SEKE'00, 2000, pp. 103–110.
[14] H.M.W. Verbeek, T. Basten, W.M.P. van der Aalst, Diagnosing workflow processes using woflan, Comput. J. 44 (4) (2001) 246–279.
[15] J.L. Woo, D.C. Sung, R.K. Yong, Integration and analysis of use cases using modular Petri nets in requirements engineering, IEEE Trans. Software Eng. 24 (12) (1998) 1115–1130.

Kees van Hee received a Ph.D. in operations research from Eindhoven University of Technology. From 1994 till 2004 he was managing partner of Bakkenist Management Consultants and Deloitte Consultancy. During 1991–1992 he was visiting professor at the University of Waterloo. From 1984 till 1994 and again since 2004 he is professor of computer science at Eindhoven University of Technology. He published articles and books on the following topics: Markov decision processes, Applications of queuing theory, Decision support systems, Formal specification methods and tools, Petri nets, Database systems and Workflow management systems.

Natalia Sidorova is an Assistant Professor at the Architecture of Information Systems group of Eindhoven University of Technology. She received her Ph.D. in computer science from Yaroslavl State University, Russia, in 1998. Her current research interests include modeling and verification techniques for concurrent systems.

Lou Somers is currently leading the development of the embedded software for a new printer family at Oce Technologies. He received his Ph.D. in theoretical physics at the University of Nijmegen in 1984. He is part time Associate Professor at Eindhoven University of Technology in the area of software engineering. He has been involved in several European and national Dutch software projects.

Marc Voorhoeve is Assistant Professor at Eindhoven University of Technology. He received his Ph.D. in mathematics at Leiden University in 1977. He was a software engineer at Philips Data Systems in Apeldoorn until 1985. His current teaching and research interests include modeling and verification of processes.