Use Cases as Workflows

Michel Chaudron, Kees van Hee, and Lou Somers

Eindhoven University of Technology, Dept. Math. & Comp.Science, P.O. Box 513,
NL-5600 MB Eindhoven, The Netherlands
{m.r.v.chaudron, k.m.v.hee, l.j.a.m.somers}@tue.nl

Abstract. In requirements engineering we have to discover the user requirements and then we have to transform them into precise system specifications. There are two essential aspects to be modeled: the data aspect and the process aspect of the system. There are many techniques available to describe these aspects but it is always difficult to integrate these views in a consistent way. Last decade two techniques are used frequently in requirements engineering: use cases and workflow models. We show that these techniques can be integrated in a natural way, using the framework of colored Petri nets. We only sketch the underlying formal framework and focus on the practical application of the approach by a case study.

1 Introduction

Requirements engineering is a distinguished field in software engineering since many years (see e.g. [16]). There are two famous problems: one is to bridge the gap between informal requirements and formal specifications, the other one is to integrate models that describe different aspects of a system. Informal requirements are important for the principals and the potential users of a system, formal specifications are essential for the software constructors. The last decade *use cases* are used more and more as a way to describe requirements. It seems that non-experts understand them better than for instance data models and dataflow diagrams. There is no formal definition of a use case that is accepted by a large community of software engineers. It is generally understood that a use case describes a “piece of functionality” of a system from the viewpoint of an actor who will work with the system. A use case combines in fact a data and a process perspective. In the early stages of system development, where the concept has to be “sold” to the principals and potential users it might be an advantage that use cases do not have a formal definition: this gives us some freedom to apply the concept and to interpret a specific use case. However, for the following phases this lack of precise semantics is a source of problems for the software engineers. Therefore, it is essential to have formal semantics for use cases. We use workflow theory for this purpose.

In the field of business process design the use of modeling techniques with a precise semantics has proven to be very useful: errors and missing parts become clear in the early stages and it saves work later on. The availability of techniques for verification of models turned out to be very valuable in practice [2]. Another term for business process is workflow. There are several formalisms to model workflows, one very successful one is to describe workflows as a special class of Petri nets, the so-called *workflow nets* [4].

In this document we apply results of workflow theory to model use cases. In fact, we consider use cases as workflows and we show how the field of requirements engineering can profit from the results of the field of business process modeling. A nice coincidence is that the term “case” occurs in workflow theory for “job” to be handled and so a workflow can be seen as a *case type*. Therefore, a use case is modeled as a case type in the sense of workflows.

The second problem is the integration of models that describe different aspects of systems. There are many articles about this subject. The use of colored Petri nets as a framework for integration has proven to be successful (see for instance [9] or [12]). Here we use the same approach, although we simply say: use cases could be modeled as workflow nets.

The remainder of this paper is organized as follows. In Section 2 we consider the place of requirements in the life cycle of a system and the steps to take to describe the functional requirements for a software engineering project. So non-functional requirements (availability, performance, adaptability, portability and many other “abilities”) are not covered in this paper. In Section 3 we consider the modeling concepts and in Section 4 we summarize the most important construction techniques. In Section 5 the main course is served: a case study where we show how the proposed approach should work.

2 Requirements Engineering in the Lifecycle

There are many standards for phasing the lifecycle of a software system. Rational Unified Process (RUP [13]) is one newest leaves of this tree. We have chosen here the standard [17] of the European Space Agency (ESA). The reason for this is that it is a well-documented standard, that the standard is used frequently in practice, and that last but not least we adopted this standard some years ago for our students in computer science. The choice of a standard for our purpose is not essential: they all distinguish requirements in some form.

The ESA standard distinguishes the following phases in the lifecycle of a software system: user requirements, software requirements, architectural design, detailed design and production, transfer, operations and maintenance.

Here, the first two phases refer to requirements. We refer to these phases together as the “requirements phase” with two sub-phases: “user requirements” and “software specification” because here the main course is the formal specification of the system.

Each phase has some deliverables as output: the product documents that determine certain aspects of the system. We do not consider the strategy to perform the phases, so our contribution is applicable to *sequential* (or waterfall), *iterative*, *incremental*, or *time boxing* strategies. In this paper we will concentrate on the production of software requirements and because this phase is in between the user requirements and architectural design, we touch these topics as well. The user requirements have a certain overlap with the software requirements, but they are written in the language of the user. The software requirements are meant for software engineers and they contain much more details. In our approach we promote the use of formal methods in the early stages because these methods encourage us to be precise and enable us to use verification methods. In this way we are able to discover inconsistencies and errors in a very early stage, which will pay off in later phases: the later an error is found the more expensive it is to correct it.

In the software requirements phase we concentrate on the functionality of the system to be made. We encourage having a complete *logical* model of the software in this phase. (We use here a pragmatic definition of the term "complete": a model is complete if it contains enough details to build or generate a computer model of the system.) A logical model of a system is often considered as a formal specification of a system. In the architectural design this logical model is translated into the specifications for the *real* system. In both phases we distinguish "components". In the software requirements the system is one logical component or it may be refined into several communicating logical components. In the architectural design the components are physical components: pieces of software, implemented on hardware. It is necessary to map the logical components to the physical ones. Sometimes it is possible to have a one-to-one mapping. However, in many cases one logical component is distributed over several physical components. It also occurs that several logical components are realized by one physical one.

The activities to be performed in the first two phases are as follows. Some terms are not explained here but they will be in the next sections. In the user requirements phase one has to:

1. Identify the *stakeholders*: all relevant roles that have some interest in the system.
2. Identify the *actors*: all stakeholders and other systems that interact with the system.
3. Describe (informally) the *use cases*: the logical "pieces of functionality".
4. Describe the *non-functional* requirements like performance or portability. This includes development constraints like the development environment or the execution platform.

In the software requirements phase one has to:

5. Transform the use cases into *workflows* (WF-nets).
6. Make a *class* model.
7. Describe class *lifecycles* (workflows).
8. Describe the *interactions* between all workflows.
9. Connect the workflow *transitions* and the *methods* in the class diagrams.
10. Perform logical system *decomposition* (if necessary).

Steps 5 and 6 may be performed in reversed order. In fact, all these tasks are performed iteratively, because in tasks 8, 9, and 10 we may discover inconsistencies that have to be repaired by redoing earlier tasks. In the software requirements phase also requirements in the form of necessary or desired properties may be formulated in terms of logic. In addition, non-functional requirements such as timeliness may be formulated in a formal way in this phase. However, we concentrate on the functional requirements here. The results of the first steps are described in the User Requirements Document (URD), the others in the Software Requirements Document (SRD). The URD roughly describes the problem the stakeholders need to address, whereas the SRD describes the specification of the proposed solution.

3 Modeling Framework

In this section we introduce the modeling concepts that are suitable to perform tasks 5 up to 10, i.e. we consider the formalisms with which we model the various aspects of a system. It is popular to use the modeling techniques of the Unified Modeling Language (UML) (see [7] for a concise introduction). This framework consists of many useful modeling techniques. In fact, we only use uses cases, class models and (a variant of) activity diagrams from UML. This does not imply that we abandon the others, but we focus here on the integration of only a few of the modeling techniques.

The things we have to model of a system are in fact the *state space* and the *events* that may change the states of the system. A state of system is defined by a set of *data objects* modeled by means of a *data* model. For events we use a *process* model. (At the end of this section we will establish the relationships between the events and data objects.)

Persistent data objects are modeled by the *class* model of UML, i.e. data objects that remain in the system if there are no events. The UML class model also allows us to define operations on the data, the *methods*. The *volatile* data that are produced and consumed during the events can be modeled with a standard *type system* or (if it is complex data) with a language like XML.

For the *process modeling* we use (high level) *Petri nets*. (For an introduction, standard terminology and many important properties of Petri nets we refer to [15] and [6].) Events are modeled by (the firing of) *transitions* and states are modeled by the *marking* of places. Petri nets are very close to UML *activity diagrams,* but have better defined semantics and many possibilities to verify behavioral properties. An alternative could be to use a *process algebra*.

The *process models* occur in two forms in our specifications: as use cases and as lifecycles for object classes. We require that these processes are a special kind of Petri nets: *workflow nets* [4]**.** Workflow nets have one initial and one final place and each node of the net lies on a path from the initial to the final place.

An additional requirement is that the workflow nets are *sound* (c.f. [1]), i.e. each reachable state (or marking) of the net lies on a path from the initial to the final state in the state space, where the initial state has only one token in the initial place and the final state has only one token in the final place. The reason for requiring that all use cases and lifecycles are sound workflow nets is that this guarantees that all use cases and lifecycles are proper *transactions* of the system: with a start and an ensured end.

The *transitions* in a use case model are *activities* of the actors that are involved in the system: users or other systems. Transitions can also be *autonomous*, which means that they are executed by the system itself as soon as the transition is enabled. In many situations the transitions are *triggered* by users in a user interface.

Tokens in the workflow nets that model use cases, represent the *case* that is handled. If there is concurrent behavior possible, then a case is represented by more than one token. Then we assume that each token has a (case) identity and that transitions, that consume more than one token at the same time, only consume and produce tokens with the same identity. Besides a case identity, a token may carry some message (volatile) data. Therefore, the tokens may have values, which means that we work with colored Petri nets [11].

We often have some global variables that are used in transitions. They are modeled by *global stores*, i.e. places that always contain one token and that are connected to many transitions by a read and write arc. (We normally do not draw these arcs to global stores in the diagrams.) The instances of a class model may be stored in a global store. Therefore, we may associate sub-class models to global stores. In fact, different global stores may have the same sub-class models.

Since we model with colored Petri (or workflow) nets, we obtain the relationship between data objects and events in a natural way: the states are modeled by the places marked with colored tokens (which are the data objects) and the events are modeled by transitions that change the states by consuming and producing colored tokens.

The last concept we need is the t-workflow. This is a Petri net with an initial and final transition instead of initial and final place. It is easy to transform one into the other: put a place (transition) in front of the initial transition (place) and a place (transition) after the final transition (place) and one obtains the other form of workflow. In addition, the concept of soundness can be translated for t-workflows: a t-workflow is sound if and only if its transformation is a sound workflow net.

4 Construction Techniques

There are several ways to build Petri nets in a structured way (see e.g. [8]). Here we do not present an exhaustive list of construction techniques; we only present some important techniques for the construction of workflow nets.

For workflow nets, it is essential that the workflows we model are sound. Non-sound workflows usually have serious modeling errors. Soundness is not the only property worthwhile to be verified, but it is the most important domain-independent "sanity check" for models. There are efficient techniques for verifying the soundness of workflows [18]. So one can model a workflow and check afterwards whether it is sound or not automatically. We have good experience with an approach that guarantees soundness by the way we construct the models: "soundness by construction".

Essentially, there are two approaches to construct workflow nets in a systematic way:

1. *Top down*: by stepwise refinement of places and transitions in a given start net.
2. *Bottom up*: by connecting existing component nets.

In practice, these approaches may be combined.

For the top down approach we have only one transformation rule: *replacement*. We start with a given set of basic workflow nets or *workflow patterns* for which we have a proof of soundness. Then we refine or replace a place by a sound workflow net or a transition by sound t-workflows. (There are some difficulties if a sub-workflow can be triggered two or more times concurrently, cf [10]). In most cases the result is a sound workflow again.

In the bottom up approach we also need only one basic transformation rule: *fusion* of places or transitions, i.e. two places (transitions) are "glued" together to become one place (transition). We have some standard constructions, sometimes using some auxiliary building blocks (also Petri nets) to assemble workflow nets: *sequential, alternative, parallel* and *iterative* composition. In sequential composition we fuse the final place (transition) of the first (t-)workflow net with the initial place (transition) of the second (t-)workflow net. In alternative composition, we fuse the initial places and the final places of two workflow nets. For the alternative composition of t-workflows, we need some auxiliary blocks: an or-split and an or-join (see Fig. 1). The parallel composition of t-workflows is simply the fusion of the initial and final transitions of the two t-workflows. For the parallel composition of ordinary workflow we need two auxiliary blocks: an and-join and an and-split (see Fig. 1). For the iterative composition we need another auxiliary building brick: a sequence of two places with a transition in the middle. The first place is fused with the final place of the workflow net and the last place of the building block with the first place of the workflow net. To make a correct workflow net of this we use this auxiliary block twice: one in front of the iterated net and one at the end. In case of a t-workflow, we use a similar construction.

There is one other important standard construction: *asynchronous coupling* between two workflow nets. In this case, we use an auxiliary building block, which is a sequence of two transitions with a place in the middle. We fuse the first transition with some transition of one of the two workflow nets and the last transition with some transition of the other workflow net. Note that in this way we may loose the workflow structure and even if we have a connected them such that the overall net is again a workflow net, then it is not sure that it is sound. However, there are some constructions that guarantee soundness (see [5]). Besides the asynchronous coupling, we have the *synchronous coupling*: where we just fuse transitions of two workflow nets, not being the initial or final ones.

Synchronous and asynchronous couplings are used frequently when we interconnect use cases and lifecycles. Synchronous coupling is applied if two events in different use cases are in fact the same. This construction suffers from the same risks as the asynchronous one concerning soundness. Asynchronous coupling is used if two workflows do not have overlapping transitions but still need some coordination: a transition of one workflow may only execute if a transition of another workflow has executed before. In a synchronous coupling transitions have to execute simultaneously, whereas in an asynchronous coupling transitions have to execute in some order. In step 8 (see also section 5.4) of the requirements phase we apply these couplings. There we use a notation technique where we list the transitions per use case and lifecycle and where we relate them (by an arc) to transitions of other use cases or lifecycles. The relationship has a direction if one of the transitions is taking the initiative: one transition is *triggering* the other. If each of the related transitions may take the initiative, or if they have to execute simultaneously there is un undirected relationship. The choice for synchronous or asynchronous coupling may be delayed, sometimes even to step 10.

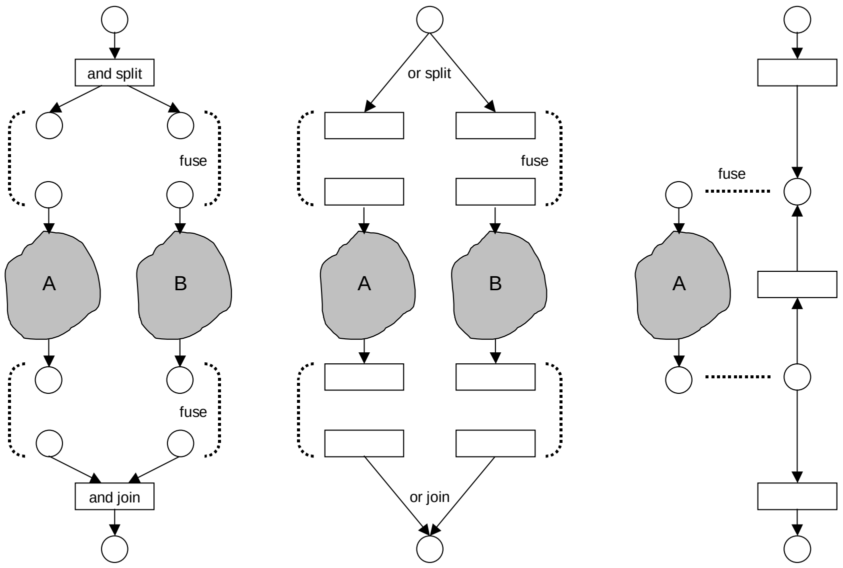

Fig. 1. Two constructions to compose workflows A and B. Iteration of workflow A

There are two other constructions we like to mention: the use of *global stores* and of *global transitions*. As mentioned before, a global store is a place that always contains one token and that is implicit (i.e. without drawing arcs) connected to a set of transitions with one consume and one produce arc. So the global stores do not influence the process flow of our workflow nets, but they are used to store variables that are shared by different transitions. Typically they are used to store a set of objects (the instance) of one class. Global transitions are implicit connected to a set of places (called "superplace") and if they are enabled by normal places, they consume all tokens available in the superplace. This can be used to enforce soundness of a workflow net and it is used frequently to model exception handling: if some event occurs we have to cancel the whole transaction. (Note that it is not always possible to simulate the behavior of a global transition without using inhibitor arcs.)

5 Case Study: The Web Shop

To illustrate the concepts treated in the previous section, we will develop the requirements for a web shop. For a part, a web shop is an ordinary shop, with items that are stored in warehouses, can be purchased by customers, are paid for, and are shipped. We will use the shopping cart metaphor, where a customer puts the products he wants to buy in a cart. In our web shop, a customer buys products that can be configured according to a certain model. So a model is a type of product. Computers or cars are typical examples of products in our web shop, but also holiday trips where the configuration is in fact the trip design. In this case study we focus on configurable physical products.

The actors are the persons, organizations, or systems that interact with the system we are going to build. In the web shop example, these might be a customer (browsing, buying, monitoring, feedback), inventory control (back ordering, shipping), the system administrator (back-up, upgrades), controlling (billing), marketing (pricing, product profiling, changing product portfolio), or design (page layout, styling).

5.1 Use Case Workflows

A use case corresponds to a task the system has to fulfill. Each use case involves a number of actors. We will show the workflows of a number of customer related use cases for the web shop and the rationale for the choices that have been made.

Use case “customer walks shop”. As a first example, we will consider the use case “customer walks shop”. The corresponding workflow is displayed in Fig. 2. Here the customer browses through the different models, configures products, adds products to his shopping cart, and possibly removes products from his shopping cart. Note that we might have modeled these steps also as four different use cases.

In the workflow definition of Fig. 2 we have constrained the cart manipulation: once the customer has selected a model, he has to put a product belonging to this model in his shopping cart, or he has to deselect it before he can view the contents of his shopping cart. A possible solution would be to define two screens allowing the customer to do both at the same time: we introduce parallel workflows. This is shown in the workflow at the left-hand side of Fig. 3. The matching of the input tokens by the exit transition is done on the case identity of the token.

Another issue is the fact that it would be more realistic to allow an exit from every state (since the customer may leave the site at any time). This is shown in the workflow at the right hand side of Fig. 3. Note that the exit may be an explicit user action, or might also be triggered by a timer event. It would be best to indicate explicitly who initiates each action (which actor or the system itself).

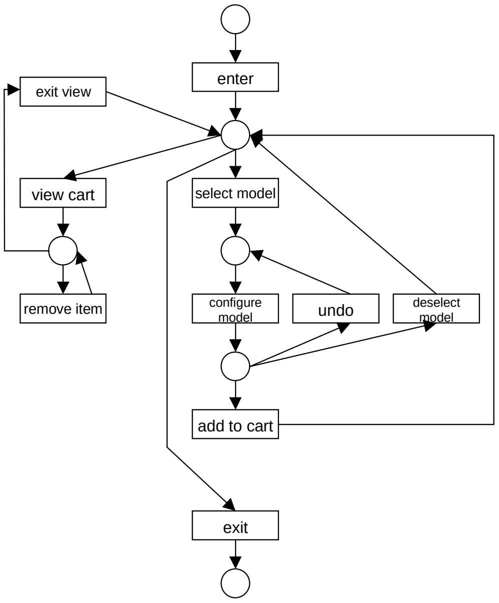

Fig. 2. First attempt to model the workflow of the use case “customer walks shop”

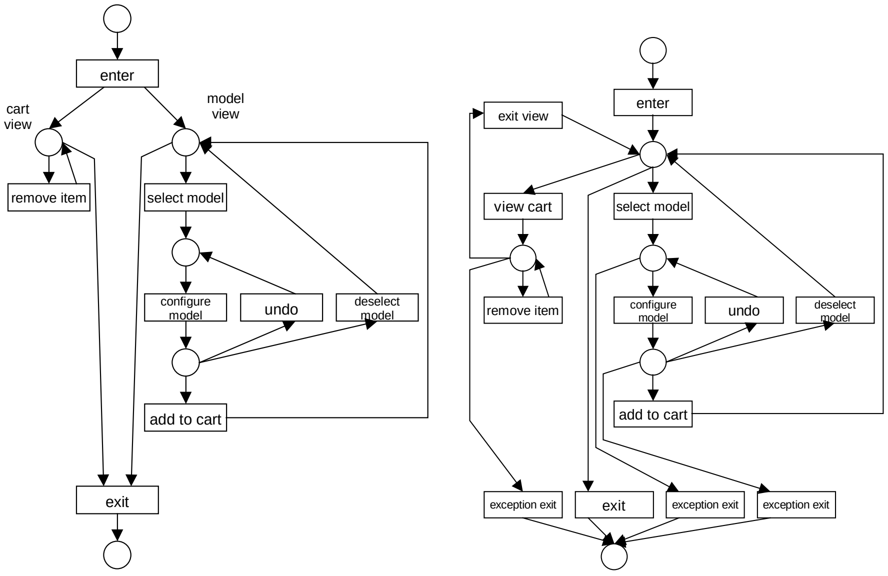

Fig. 3. Two alternatives for the use case “customer walks shop”. The first one allows the customer to configure products and to view and manipulate the contents of his shopping cart at the same time. The other one allows an exit at any time

Use case "customer buys products". Another example is the workflow corresponding to the use case "customer buys products". Here the customer has a non-empty shopping cart and wants to buy these items. Two equivalent models are shown in Fig. 4. We have used a "super place" to model the exceptional exit flows.

For each choice of payment method (inter bank, credit card, cash on delivery, or paycheck), some customer solvability check has to be applied. The inner workings of these workflows may be different and will show up once we try to detail the transition "check customer solvability". We might also add activities to check the existence of the customer data and to add newly filled in data into the customer administration.

Use cases "back office" and "handling". The "back office" use case controls the work after the customer has agreed upon buying a product. The workflow is shown in Fig. 5. Only for pay on delivery, we have modeled the possibility that the customer does not pay.

We will use a simple version of the use case "handling" in which the ordered products are assembled and shipped. Note that for such workflows (like the shipping operation) some standard patterns exist in the literature.

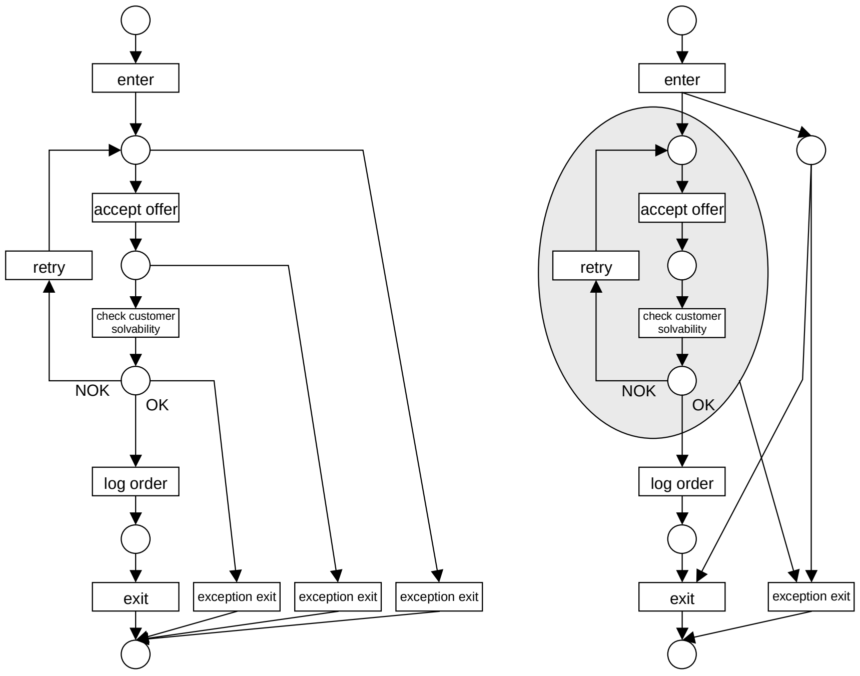

Fig. 4. Use case "customer buys products", variants without and with a superplace and a global transition ("exception exit")

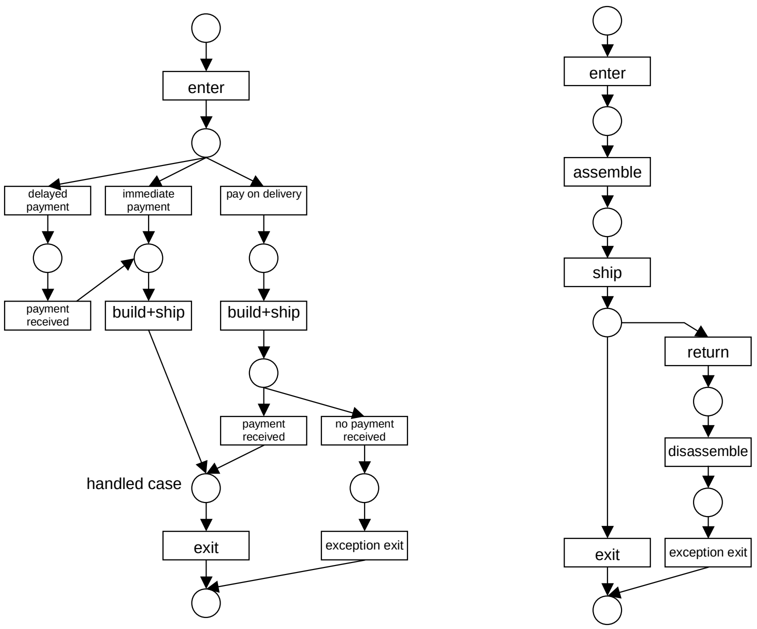

Fig. 5. Use cases "back office" and "handling"

5.2 Class Diagram

We use an adapted and extended version of the class diagram pattern for internet shops of [14]. It only models the classes involved in the customer related use cases. The elaboration of the payment and billing part of the data model is not treated.

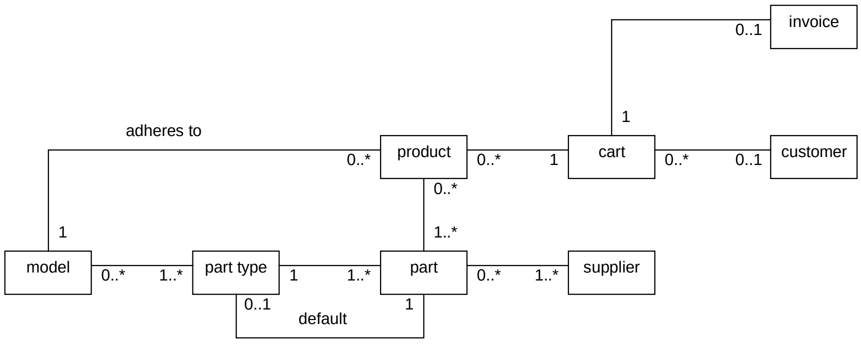

Fig. 6. Class diagram of the web shop

Each visit of a customer to the web shop involves a shopping cart. For "product" one might think of e.g. a PC configured with a number of components (parts). Each product adheres to (is configured according to) a model. Such a model has part types (like a hard disk) that may be chosen from a number of allowed parts. Of course, a configured product may only contain those parts that are allowed by the part types of its model. In the table below, we list some attributes.

Table 1. Classes and some attributes

Class	Attribute
cart	payment mode
product	amount
	color
	price
model	assembly price
part	price
customer	name

5.3 Class Lifecycles

Each instance of a class in the class diagrams has a lifecycle. We will also use workflow nets for these lifecycles. Note that each transition in a class lifecycle is usually caused by an external event.

Many lifecycles are very simple, as shown by the examples of Fig. 7. Note that once a model has a relation with a product, one is not allowed to change it anymore. Many products may use the same model at the same time, and an update of such a model is only allowed if all products have released the model. This is modeled by having n (an arbitrary large number) tokens in the central state of the model lifecycle.

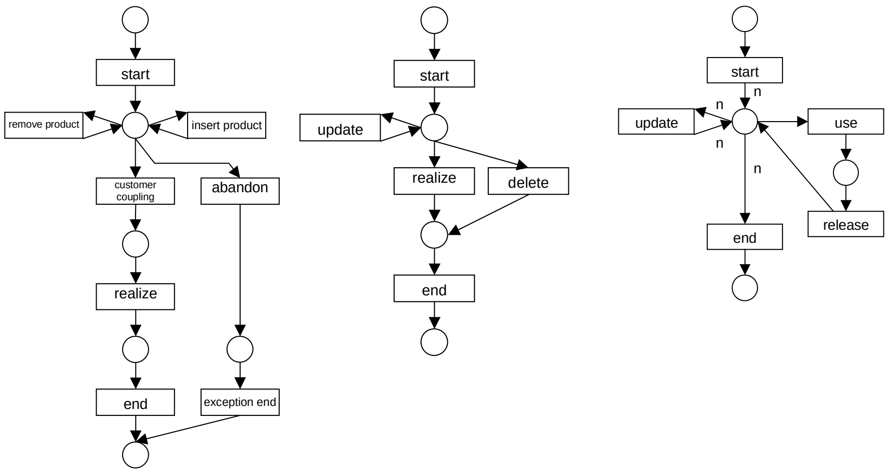

Fig. 7. Lifecycles of shipping cart, product, and model

5.4 Interactions between Workflows

The next step is to associate transitions of the use case workflows and the class lifecycles. In Fig. 8 we show how the transitions are associated.

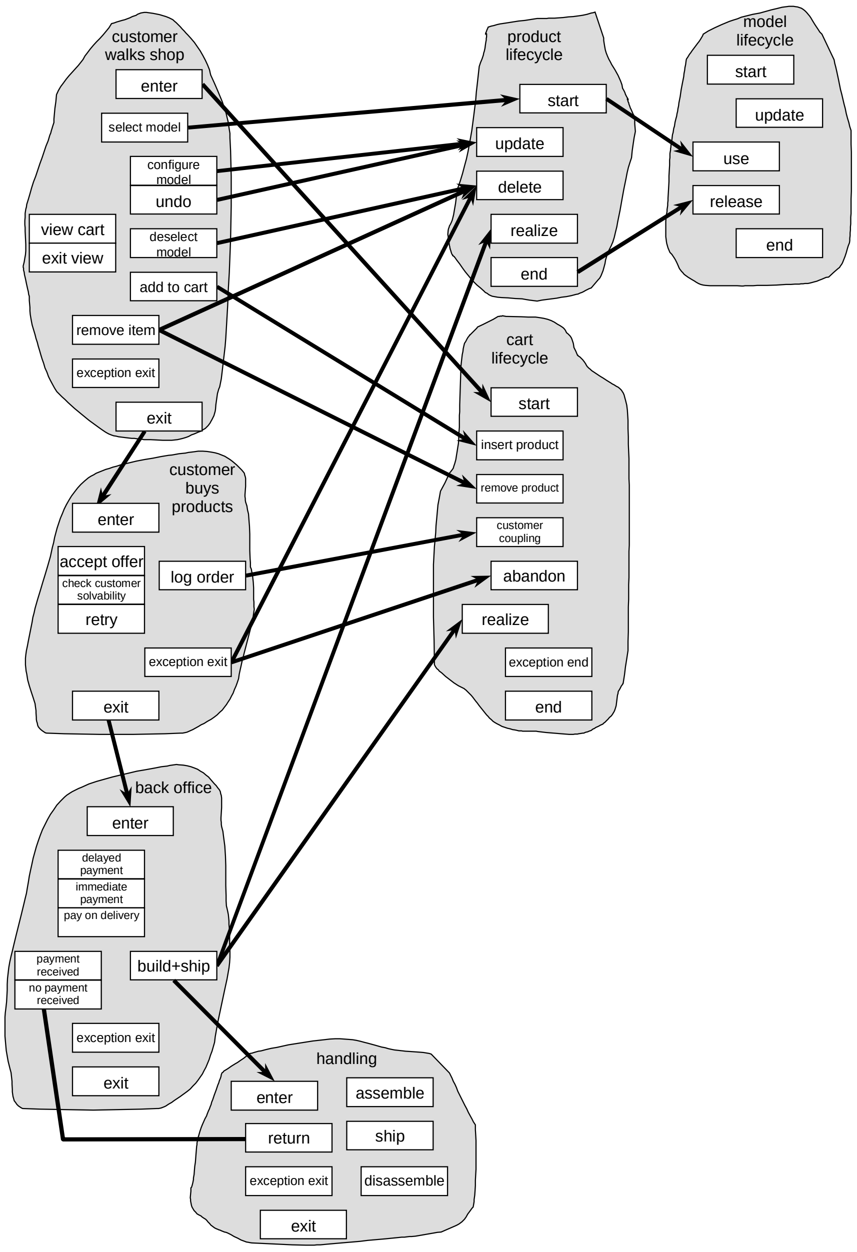

Fig. 8. Relations between workflow transitions. Use case transitions are shown on the left, life cycle transitions on the right

Relations between transitions in two use cases. A transition in a use case can start another use case. This may be the exit transition like in "customer walks shop" that starts (upon normal termination) the workflow "customer buys products". It can also be any other transition: for example, "build+ship" in "back office" starts the workflow "handling". This coupling is usually asynchronous.

A transition in a use case can also trigger a transition in another use case. For example, "return" in "handling" fuses with the "no payment received" in "back office". Here we must fuse the transitions to guarantee that "return" is only allowed to fire in case of pay on delivery.

Relations between transitions in a use case and a lifecycle. Now we look at the mapping from use case transitions to lifecycle transitions. This mapping can be synchronous or asynchronous, but will probably never be realized by transition fusion: the transitions in a data object can be triggered by many use cases.

A transition in a use case may be associated to multiple lifecycle transitions, each in a *different* lifecycle. For example, "remove item" in "customer walks shop" means that both "delete" and "remove product" should fire.

Multiple transitions in the same or different use cases may also be mapped to one lifecycle transition. For example, "deselect model" and "remove item" both mean that "delete" should fire. However, this coupling has a direction: if "delete" fires, this does not imply that also "deselect model" should fire.

Another reason for not having a one-to-one mapping might be that the modeling has been performed at different levels of abstraction: a transition corresponds to a subnet. This does not occur in our example.

Note that not all transitions of a use case have to take part in a mapping. For example, the "view cart" transition in the "customer buys items" workflow only retrieves the current contents of a shopping cart and therefore does not take part in the lifecycle of the cart.

Relations between transitions in two lifecycles. Usually, we will also have relations between transitions in different class lifecycles: if a class changes state, a change may also occur in the state of a dependent class. For example, if we create a product, the corresponding model will be locked: no one is allowed to change it anymore.

5.5 Coupling Workflow Transitions and Class Methods

In the following table, we give an informal overview of the methods that are called if a transition of a specific use case fires. In a latter stage, we have to specify exactly what the input and output parameters of each method are and how they are related to the data carried by the workflow of a use case.

In the table, we see for example that the workflow "customer walks show" has two instance variables representing global data (cart and current product). Those are filled if the "start" transitions of the lifecycles of cart and product fire, which is caused by the transitions "enter" and "select model" of the workflow.

Table 2. Some workflow transitions and related method calls of lifecycle transitions.

Workflow	Transition	Class	Transition	Method call or relation change
customer walks shop: cart, current product	enter	cart	start	(creation of new instance)
	select model	product	start	(creation of new instance)
	configure model	product	update	Change product-part relation. Update attributes of product (amount, color, and price).
	undo	product	update	Change product-part relation. Update attributes of product according to default (amount, color, and price).
	deselect model	product	delete	--
	add to cart	cart	insert product	Add relation between cart and product.
	remove item	cart	remove product	Remove relation between cart and product.
		product	delete	--
	view cart	--	--	--
	exit view	--	--	--
	excep. exit	--	--	--
	exit	--	--	--
customer buys items: cart, requested payment mode	log order	cart	customer coupling	Create new customer class if not yet existing. Add relation between cart and customer class. Update payment mode attribute.
	…..	…..	…..	…..

6 Conclusions and Future Work

We have shown how workflow theory can be applied to requirements engineering, in particular for the formalization of use cases. It is again a confirmation that the colored Petri net framework is a sound base for model integration. The approach presented here gives a systematic way to develop system specifications and the possibility to verify properties, in particular soundness. We did not have enough room here to show how this approach can be continued to decompose a system into logical components, but [12] and [5] provide a theoretical base to support this approach. Experience in several software development projects has convinced us that that approach works well.

We are working on a design method for component based development where workflow nets are first class citizens, i.e. we try to model all process aspects of a system components as workflows. We like to do this using standard techniques like use cases, sequence charts and activity diagrams as much as possible. The idea is to add some modeling restrictions (as conventions) to the existing techniques and to limit the introduction of additional notations as much as possible.

References

1. Aalst, W. van der: Verification of Workflow Nets. In: Azema, P., Balbo, G. (eds.): Application and Theory of Petri Nets 1997. Lecture Notes in Computer Science, Vol. 1248. Springer-Verlag, Berlin (1997) 407–426
2. Aalst, W. van der, Desel, J., Oberweis, A.: Business Process Management, Models, Techniques, and Empirical Studies. Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin (2000)
3. Aalst, W.M.P. van der: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers, Vol. 8, no. 1 (1998) 21–66
4. Aalst, W. van der, Hee, K. van: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)
5. Aalst, W. van der, Hee, K. van, Toorn, R. van der: Component-based Software Architectures: a Framework Based on Inheritance of Behavior. Science of Computer Programming, Vol. 42 (2002) 129–171
6. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, Vol. 40. Cambridge University Press (1995)
7. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. 2nd edn. Addison Wesley (2000)
8. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer-Verlag, Berlin (2002)
9. Hee, K. van: Information Systems Engineering: A Formal Approach. Cambridge University Press (1994)
10. Hee, K. van, Sidorova, N., Voorhoeve, M.: Soundness and Separability of workflow nets. Submitted for publication
11. Jensen, K. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic concepts. 2nd corrected printing. Springer-Verlag, Berlin (1997)
12. Kindler, E., Martens, A., Reisig, W.: Inter-Operability of Workflow Applications: Local Criteria for Global Soundness. In: Aalst, W. van der, Desel, J., Oberweis, A.: Business Process Management, Models, Techniques, and Empirical Studies. Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin (2000) 235–253
13. Kruchten, P.: The Rational Unified Process: An Introduction. 2nd edn. Addison-Wesley (2000)
14. Fernandez, E. B., Liu, Y., Pan, R.Y.: Patterns for Internet shops. In: Procs. of PLoP 2001, http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/PLoP2001/ebfernandez0/PLoP2001_ebfernandez0_1.pdf
15. Reisig, W.: Petri Nets, An Introduction. Springer-Verlag, Berlin (1985)
16. Sommerville, I.: Software Engineering. 6th edn. Addision-Wesley (2000)
17. Mazza, C., Fairclough, J., Melton, B., de Pablo, D., Scheffer, A., Stevens, R.: ESA Software Engineering Standards. Prentice-Hall (1994)
18. Verbeek, H., Aalst, W. van der: Woflan 2.0: A Petri-Net-Based Workflow Diagnosis Tool. In: Nielsen, M. Simpson, D.: Procs. 21st International Conference on Application and Theory of Petri Nets. Lecture Notes in Computer Science, Vol. 1825. Springer-Verlag, Berlin (2000) 475–484