Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Formal Software Specifications with Executable Use Cases and Coloured Petri Nets

Jens Bæk Jørgensen · Simon Tjell · João M. Fernandes

Received: date / Accepted: date

Abstract This paper presents Executable Use Cases (EUCs), which constitute a model-based approach to requirements engineering. EUCs may be used as a supplement to Model-Driven Development (MDD) and can describe and link user-level requirements and more technical software specifications. In MDD, user-level requirements are not always explicitly described, since usually it is sufficient that one provides a specification, or platform-independent model, of the software that is to be developed. Therefore, a combination of EUCs and MDD may have potential to cover the path from user-level requirements via specifications to implementations of computer-based systems.

Keywords requirements engineering · requirements and specifications · platform-independent models · model-driven development · coloured Petri nets

Jens Bæk Jørgensen
Mjølner Informatics
Finlandsgade 10, 8200 Aarhus N, Denmark
Tel.: +45 70274343
Fax: +45 70274344
E-mail: jbj@mjolner.dk

Simon Tjell
Dept. Computer Science, University of Aarhus
Aabogade 34, 8200 Aarhus N, Denmark
Tel.: +45 89425600
Fax: +45 89425601
E-mail: tjell@daimi.au.dk

João M. Fernandes
Dept. Informática & CCTC, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
Tel.: +351 253604454
Fax: +351 253604471
E-mail: jmf@di.uminho.pt

1 Introduction

MDD [33] can offer significant support for the path from user-level requirements, often based on observations of the real world and informal descriptions, via specifications to implementations of computer-based systems. MDD focuses on automatically transforming software models into running implementations on various execution platforms. This implies that MDD is essentially a solution-oriented approach. MDD approaches do not always emphasise the requirements engineering activities needed to produce the necessary specifications. For example, OMG's Model-Driven Architecture (MDA) [43] does not encourage software developers to pay enough attention to properly identifying and describing the problems that the software must solve. This may result in obvious problems, like developing a perfect solution for the wrong problem.

With this observation as motivation, we propose *Executable Use Cases* (EUCs) [21], a model-based approach to requirements engineering that can be used together with MDD. Figure 1 illustrates the relationship between EUCs and MDD. The starting point for any software development project is some real-world problem for which some stakeholders have chosen to devise a solution. The analysis of this problem results in the production of some agreement on the most relevant and important requirements to a solution. The analysis should also result in a common understanding of the problem at hand. Based on the requirements, a specification is developed to more precisely describe a specific software system that solve the problem when embedded in the environment. This specification is then the foundation for developing the actual software. Figure 1 also informally shows the stages at which EUCs and MDD are most useful. This illustrates that the two ap-

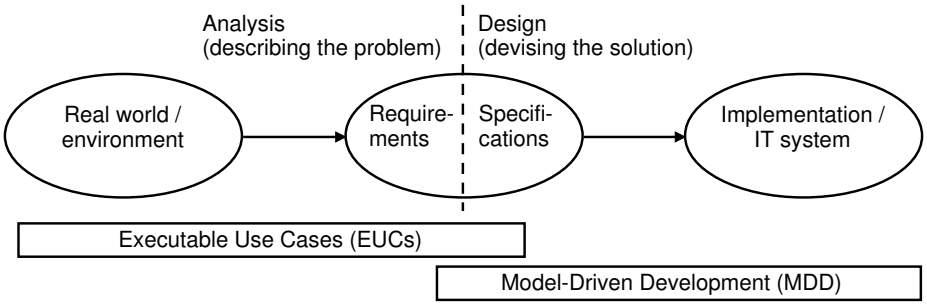

Fig. 1: Relationship between Executable Use Cases and Model-Driven Development.

proaches cover different parts of the development cycle. It also depicts the fact that there is an overlap in the areas covered by the two approaches. In this paper, we investigate how this overlap may be exploited in order to devise an approach to combining the use of EUCs and MDD.

Generally, in Software Engineering, the terms 'requirement' and 'specification' are often used with many different meanings [13,36]. The combination 'requirements specification' is also very common (e.g. [35,39]). In the description of Figure 2, we give a more restrictive definition of the terms since they are essential to the introduction of EUCs. This definition is in accordance with the terminology of Jackson [13,14,41].

A *requirement* is a desired property that we want to be fulfilled in the environment, for example, that the car reduces its speed when the driver steps on the brake pedal. Requirements belong to the users' world and do not need to mention the computer-based system and the software in consideration.

A *specification* is a description of an interaction between the environment and the computer-based system. For example when the driver steps on the brake pedal, a brake-by-wire controller computer system that is to be developed will receive a stimulus and, in response, will send an electrical signal to activate the physical brake. Specifications belong in the borderline between the system and the real-world.

The speed of the car and the driver stepping the brake pedal are examples of *phenomena* that exist in the environment. A phenomenon is typically an event or a state. Some of the phenomena in the environment are *shared* with the software system under development. When a phenomenon is shared, it means that both the environment and the system may observe it, but only either the system or the environment controls the phenomenon. For example, the brake pedal may be represented as a shared state phenomenon. The brake-by-wire controller is able to monitor the level of the pedal in order to detect when it is stepped down and how deep. On the other hand, the controller is not able to affect the state of the pedal at all. This is only possible to the driver, which is part of the environment. This distinction is very important, because it defines the limitations on which responsibilities may be assigned to the system under development.

The collection of shared phenomena form the boundary between the system and its environment as illustrated in Figure 2. Here, according to the terminology of Jackson, the machine corresponds to the system under development (e.g., the brake-by-wire controller). The environment phenomena are those controlled by the environment and not shared with the machine, while the machine phenomena are controlled by the machine and not shared with the environment. The terms *hidden* or *private* are sometimes used to refer to phenomena that are not shared. An example of an environment phenomenon that is not directly accessible to the brake-by-wire controller is the current speed of the car. This means that the speed should not be referred to when expressing the specification but could well be part of a requirement. Such a requirement could describe how fast the speed should decrease as a function of stepping down the pedal. The specification interface contains those phenomena that are shared between the machine and the environment. The shared state phenomenon representing the level of the pedal described above is a good example of what could be referred to in a specification.

The requirements should be expressed solely in terms of the environment phenomena, including the ones in the specification interface. The specifications should be expressed solely in terms of the phenomena in the specification interface. However, we sometimes allow for some *implementation bias* [13]. This means that the specification refers to abstractions of machine phenomena that are not necessarily shared in the specification interface. In this way, the solution space may be superficially explored in the later stages of the requirements analysis.

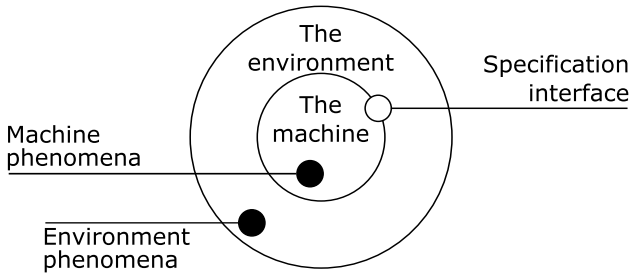

Fig. 2: Relationship between requirements and specification. Adopted from [13].

In any software development project, it is essential to pay proper attention to both requirements and specifications. On one hand, stakeholders must be involved in identifying, eliciting, prioritising, and negotiating requirements. On the other hand, software developers need specifications to have operational starting points for detailed design and even for implementation. However, often the requirements are not explicitly formulated — they just exist in the environment without being caught and written down or represented explicitly. This is in contrast to specifications, which are usually produced in plan-driven approaches. An example of a popular means to writing specifications is use cases, in the style of UML diagrams [26,44] or in textual form [6].

One important use of requirements and specifications is to give adequacy arguments for the software to be developed. For example, if one's job is to develop the brake-by-wire controller just discussed, it is important to argue that if the specification is satisfied (the brake pedal produces a stimuli to the brake-by-wire controller, which sends a signal to the physical brake, and so on), then it implies that the requirement is also satisfied (the car starts to slow down, when the driver steps on the brake pedal). A solid argument of this kind involves that we must make assumptions about how external entities in the environment work. Software engineers cannot influence or change how brake pedals work or how the physics of the car affects its speed — they are given — but it is crucial to know their properties, because the controller must interface with them. As an example, we can assume that when a brake signal is sent, the physical brakes (rapidly) become activated and cause the car to slow down. This assumption relies on a chain of causality since the brake-by-wire controller is only indirectly able to affect the speed of the car. It is the system consisting of the controller plus the brake pedal, the physical brakes, and other mechanisms that must produce the desired effect in the environment. With EUCs, we provide an approach to describe and link requirements and specifications. In particular, EUCs can be used to give adequacy arguments.

This paper extends a workshop paper [23] by a more thorough introduction to CPN, the description of combining CPN and SDs, and a more detailed discussions about the relationships between requirements and specification in the context of EUCs. The paper has the following structure. In Section 2 EUCs are presented. Section 3 describes two examples in which EUCs were applied. In Section 4, we introduce *Coloured Petri Nets* (CPN) [16,25], which has been the modelling language used by us to give formal support to our approach. The section also provides a discussion on the adequacy of this particular modelling language in the context of EUCs. In Section 5, the combined usage of CPNs and sequence diagrams is discussed, namely in what concerns the formalisation of EUCs. Section 5 presents some related work and Section 7 draws some conclusions, discusses about the relationship between EUCs and MDD, and briefly mentions related and future work.

2 Executable Use Cases

An EUC [21] supports description and validation of requirements and specifications. In a single description, an EUC can represent desired behaviour of the environment (requirements), desired behaviour of the computer-based system (specifications), and assumed behaviour of external entities in the environment (often needed in adequacy arguments).

Despite the name, an EUC can have a broader scope than a traditional use case. The latter is often a description of a sequence of interactions between external actors and a computer-based system. As noted above, a traditional use case in this way often constitutes a specification, rather than a requirement. An EUC can go further into the environment and also describe potentially relevant behaviour in the environment that does not happen at the interface. Jackson explains why requirements often need to be described in terms of phenomena found in the physical environment rather than in the interface between the system and the environment [15]. It is this property that enables an EUC to represent both requirements and specifications.

The name *Executable Use Cases* was chosen to facilitate a quick and rough explanation of the main concepts and ideas behind our approach. The stakeholders in the projects in which we have used the EUC approach have always been familiar with traditional use cases, and the essence of an EUC is to make an executable representation of what is often already described with a traditional, and well-known, use case. As can be seen from Figure 3, an EUC consists of three tiers.

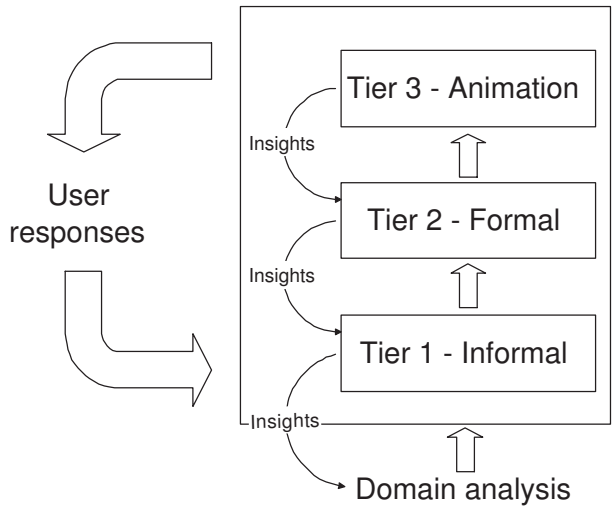

Fig. 3: Executable Use Cases.

The tiers describe the same things, but use different languages: tier 1 is an informal description; tier 2 is a formal and executable model; tier 3 is a graphical animation of tier 2, which uses only concepts and terminology that are familiar to and understandable for the future users of the computer-based system.

The three tiers of an EUC should be created and executed in an iterative manner. The first version of tier 1 is based on domain analysis, and the first versions of tiers 2 and 3 are based on the tier immediately below. Tier 1 represents typical artefacts of the requirements engineering activities, and is created routinely in many projects, often consolidated in the form of traditional use cases. Tier 1 should be the result of the collaboration among a broad selection of users, software developers, and possibly other stakeholders, with the purpose of discovering and documenting the requirements for a computer-based system.

Validation is supported through execution. This is possible at tier 2, but can only be done properly by people who are able to read and understand the formal model. In practice, this often means only software developers. However, tier 3 enables users to be actively engaged in validation by investigating the consequences of the current description as realised at tier 2. Elicitation, in the same way as validation, can be supported through execution. When users interact with tier 3, they often encounter questions, experience the EUC to behave in unexpected and maybe unsuited ways, or discover that relevant aspects have not been covered yet. In each such case, it is possible to revisit the formal model at tier 2, or even the natural language descriptions at tier 1, in an attempt to find answers to the questions raised at tier 3, and, consequently, remodel at tier 2, rewrite at tier 1, or both, to produce an improved version of the EUC.

In contrast to traditional use cases, EUCs talk back to the users and support experiments and trial-and-error investigations.

3 Examples of Systems using EUCs

In this section, we present two computer-based systems in which EUCs were applied for their development. First, we consider an Elevator Controller — an example of a reactive system — and we describe how EUCs are applied. Next, we consider the Pervasive Health-Care System (PHCS) [4], a real system aimed at use at Danish hospitals.

3.1 Elevator Controller

The elevator controller is a standard text book example; our version has been taken from [40]. The main responsibility of the controller is to control the movements of elevator cages in a high-rise building. Movement is triggered by passengers, who push request buttons. On each floor, there are floor buttons, which can be pushed to call the elevator; a push indicates whether the passenger wants to travel up or down. Inside each cage, there are cage buttons, which can be pushed to request to be carried to a particular floor. In addition to controlling the movements of the cages, the controller is responsible for updating a location indicator inside each cage that displays the current floor of the cage.

We have discussed the elevator controller and its EUCs in [18]. The CPN model itself, without the informal tier and the animation tier, has also been the subject of some papers, for example in [1].

The animation tier of the EUC, which is depicted in Figure 4 for a configuration with ten floors and two cages, represents the elevator shaft with the elevator cages, the floor buttons, the cage buttons plus the location indicator for each of the cages.

The link between the formal tier and the animation tier is that the execution of the formal tier causes drawing functions to be called. In this way, the graphical objects are animated (e.g., cage icons are moved, or location indicators are changed) in the animation tier.

Examples of requirements for the elevator controller are:

– *Collect passengers:* When a passenger pushes a floor button on floor f, eventually an elevator cage should arrive at floor f and open its doors;

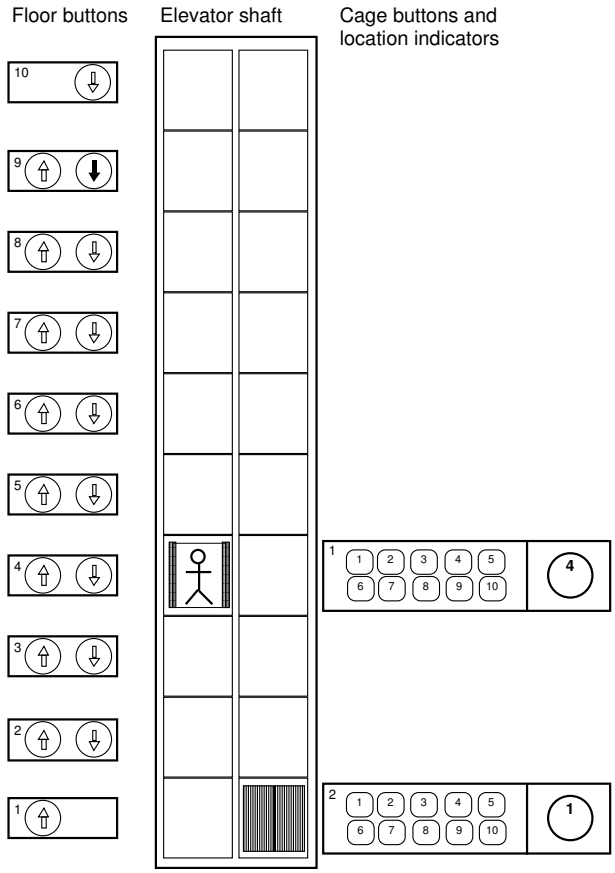

Fig. 4: EUC animation tier for the elevator controller.

– *Deliver passengers:* When a passenger pushes the cage button for floor f in an elevator cage, eventually the elevator cage should arrive at floor f and open its doors;
– *Show floor:* When a cage arrives at a floor, passengers inside the cage should be informed about the current floor number.

We now consider a specification related to the *Collect passengers* requirement.

1. Assume that a floor button is pushed;
2. The controller must receive a stimulus from the floor button;
3. The controller must turn on the light of the pushed button;
4. The controller must allocate the request to one of the cages. In particular, this implies that the controller must determine whether the request can be served immediately. This is possible only if the request comes from a floor where currently there is an idle cage. In this case, the cage can just open its doors; it is not necessary to start the motor;
5. If it is necessary to start the motor, the controller must generate an appropriate signal to the motor;
6. If it is sufficient to open the doors, the controller must generate a signal to the doors instructing them to open.

The EUC describes this scenario and its continuation; it also describes many other scenarios. The formal tier describes precisely a number of interactions between the elevator controller and external entities like buttons, sensors, motors, and doors. In the animation tier, only the consequences of the technical specifications are visible in terms of the resulting behaviour of the physical entities.

A user can push floor and cage button icons in the animation tier. For each push, the user experiences that the animation eventually shows an elevator cage icon with open doors at the requested floor, and that the location indicator icons are properly updated during the emulation of elevator movement.

When this happens, the animation tier is used to validate that the current specification of the controller and the modelled environment properties together ensures that the requirements are fulfilled, for the considered scenarios. This is the adequacy argument that we are pursuing. The graphical animation can also be used to discover problems, both simple ones (like an elevator cage, which does not stop if it comes to a floor for which it has a request) and more complex ones (like the scheduling not being done so that efficient use of the elevator cages is ensured).

However, the animation tier (tier 3) cannot be used to investigate the causes of and ultimately find solutions to the problems. For debugging, it is necessary to inspect the more technical description of the specifications that is found at the formal tier (tier 2).

3.2 Pervasive Health-Care System

In contrast to the elevator controller, this section describes how EUCs have been applied in a real-world project: the Pervasive Health-Care System (PHCS). The objective of the PHCS is to ensure smooth access to and use of hospital computer-based systems by taking advantage of pervasive computing. The PHCS is context-aware, which means that it is able to register and react upon certain changes of context. More specifically, nurses, patients, beds, medicine trays, and other items to be found at hospitals are equipped with radio frequency identity (RFID) tags, enabling presence of such items to be detected automatically by involved context-aware computers, for example, located in the medicine cabinet and in the patient beds.

Another property of the PHCS is that it is propositional in the sense that it makes qualified propositions, or guesses. Context changes may result in automatic generation of buttons, which appear at the task-bar of computers. Users must explicitly accept a proposition by clicking a button — and implicitly ignore or reject it by not clicking. The presence of a nurse holding a medicine tray for patient P in front of the medicine cabinet is a context that triggers automatic generation of a button `Medicine plan:P` on a context-aware computer located in the cabinet, because in many cases, the intention of the nurse is now to navigate to the medicine plan for P that specifies the medicine that must be poured for P. If the nurse clicks the button, she is logged in and taken to P's medicine plan.

We have used an EUC to represent the work process in medicine administration, covering nurses' pouring and giving of medicine. The EUC describes how medicine administration is supposed to be supported by the PHCS. The use of EUCs in requirements engineering for the PHCS is described in detail in [19,20].

The animation tier of the EUC, which is shown in Figure 5, represents a hospital department where nurses are walking around, pouring medicine, and giving medicine to patients. It also shows context-aware computers and their reactions to changes in the context and to the interactions of the nurses with them.

Examples of requirements for the PHCS are:

- *R1* — *Find plan*: In the medicine room, any nurse should be able to quickly find the medicine plan for any of her assigned patients.
- *R2* — *Ensure confidentiality*: When a nurse leaves the medicine room, no sensitive patient data must be left for public viewing (data must be kept confidential).
- *R3* — *Access data*: In the medicine room, it should be possible for any nurse to access the record for any of her assigned patients.

Notice that these requirements are genuinely independent of any technologic property that should be satisfied, no matter if the patient records are on paper, are only accessible electronically via a desktop-based patient record computer system, are accessible via PDAs, are accessible via the PHCS, or are made available through some other means. The requirements seem to be quite stable; it is likely that *R1*, *R2*, and *R3* are also valid requirements for a new hospital system, say, in five or ten years. In contrast, solution proposals — specifications — are more volatile. This is, in its own right, an important argument for explicitly distinguishing between requirements and specifications.

Examples of specifications for the PHCS are:

- *S1*: When a nurse approaches the medicine cabinet, the medicine cabinet computer must add a login button and a patient list button for that nurse to the task-bar.
- *S2*: When a nurse leaves the medicine cabinet, if she is logged in, the medicine cabinet computer must blank off its display, remove the nurse's login button and patient list button from the task-bar, and log her out.
- *S3*: When a nurse selects her login button, she must be added as a user, and the login button must be removed from the task-bar of the computer.

We have used the EUC to give adequacy arguments that link requirements and specifications. For example, the EUC relates the satisfaction of requirement (R1) to specification (S1). When the user interacts with the EUC through the graphical animation, he experiences that when it is emulated that a nurse enters the medicine room, the medicine plan of any of her assigned patients can appear on the display of the medicine cabinet computer icon with just two clicks; first on the patient list button, and then on the name of the patient of concern. Thus, if a computer-based system is constructed that meets (S1), and it has a reasonable performance, (R1) is satisfied. Similarly, the EUC links requirement (R2) and specification (S2); and requirement (R3) and specification (S3), respectively.

4 CPN as EUC Formal Tier Language

In our presentation of the EUC approach in Section 2, we did not fix the language to be used at tier 2, the formal tier. There are different possible choices. We could for example use a suitable programming language or a general, graphical modelling language such as statecharts [11], UML state machines or activity diagrams [44], or Petri nets [30]. These languages differ in a number of ways, and in particular they have different degrees of formality and rigidity and distinct tool support.

In our use of EUCs so far, and in particular in the industrial projects we have been involved with, we have used the formal modelling language *Coloured Petri Nets* (CPN) [16,25] as the tier 2 language. We have chosen CPN because we have experience with this language, but more importantly because CPN is appropriate for EUCs, as we argue in Section 4.3, and its tool support is quite good.

4.1 An Introduction to CPN

CPN is one out of many modelling languages in the family of languages based on Petri nets [30]. The for-

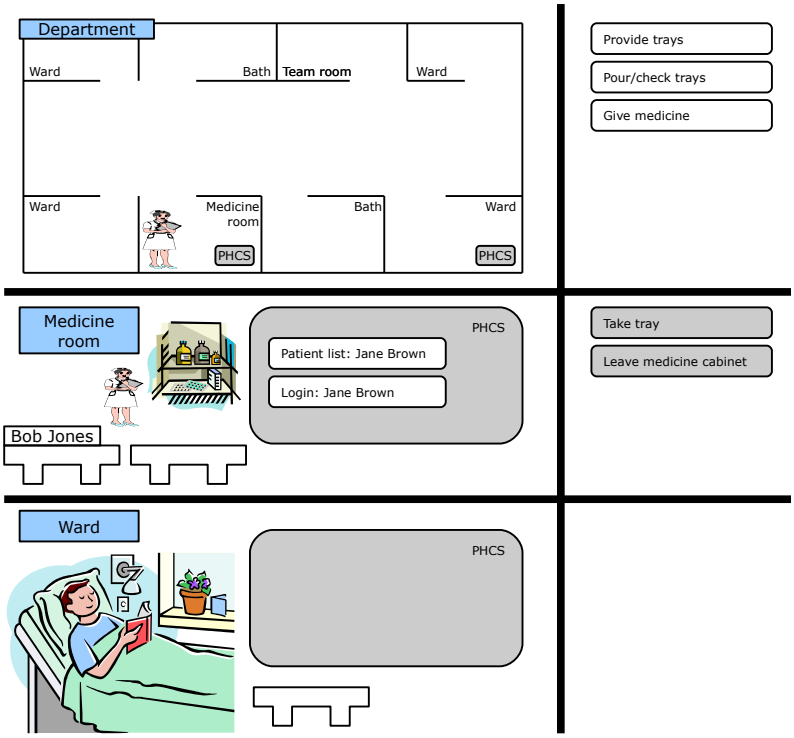

Fig. 5: EUC animation tier for PHCS.

malism behind Petri nets was originally defined in 1962 by Carl Adam Petri in his doctoral thesis [31].

What is commonly understood by a Petri net model is a mathematical structure with a graphical representation. A model is composed by a collection of basic syntactical components: *places*, *transitions*, and *arcs*. These components are graphically represented as ellipses, rectangles, and directed arcs respectively.

Places hold collections of *tokens* and thereby represent local states (*markings*). The global state of a model is represented by the distribution of tokens throughout its places. The places have an *initial marking* representing the initial local state.

Arcs lead either from a place to a transition or the other way, but never between two places or two transitions. In the first case, the arc enables the transition to remove (or *consume*) one or more tokens from the place. In the second case, the arc allows the transition to *produce* tokens in the place. The consumption and production of tokens to places occurs when transitions are *fired* during the execution of a model. At each step of such an execution, one or more transitions may be *enabled*, i.e., ready for firing. A transition is enabled if it is able to consume a specified collection of tokens from its *input places* (those connected to the transition by incoming arcs). If at least one transition is enabled, one of the enabled transitions can be chosen for firing and the execution is able to proceed to possibly performing the next step based on the new markings of the places.

Two or more transitions may be in *conflict* if they are enabled in the same step. This occurs if there is an overlap in the collections of tokens on which they depend for their enabling, i.e., if the firing of one transition results in the other one being no longer enabled in the following step. If two or more transitions are not in conflict in a given marking, they can be considered as truly concurrent and may be fired in any arbitrary order resulting in the same global state. This property is known as the *diamond rule* [5]. For formalisms where tokens have values (such as CPN), the diamond rule holds at the level of binding elements [17]. It is related to the rules about *locality* in the net structure: the enabling of a transition only depends on the marking of the input places to the transition and the result of the firing of a transition is only observable through the marking of the output places of the transition.

Up to now, we have described the issues that are common to almost all classes of modelling formalisms based on Petri nets. One of the basic points at which classes of Petri nets differ is in how much information is represented by the marking of places. Bernardinello et al. identify three levels of Petri nets with respect to the marking of places [2].

At level 1, places have a Boolean marking meaning that they either hold zero or one token that does not represent a value. All arcs consume or produce exactly one token. This level corresponds to the principles of the formalism originally defined by Petri. Elementary Net Systems [37] also belong at this level.

At level 2, places hold an integer number of tokens that are anonymous, which means that one token is not distinguishable from another. The arcs may have *weights* indicating an integer number of tokens to be consumed or produced. Graphically, the weights are represented as annotations to the arcs. An example of a formalism at this level are Place/Transition systems [32].

Finally, at level 3, tokens have values of primitive or complex data types (e.g., integers, text strings, records). Instead of weights, the input arcs have inscriptions, i.e., expressions specifying the constraints on the collection of tokens that should be consumed. Output arcs may contain expression describing the collection of tokens that is produced. In this way, it is possible to model the selection and manipulation of data in the model. In addition, transitions may contain *guards*: boolean expression over the values of tokens in the input places that must evaluate to true for the transition to be enabled. CPNs constitute an example of a formalism belonging to this level.

We often refer to nets at Levels 1 and 2 as *low-level nets* and to nets at Level 3 as *high-level nets*. In addition to these basic principles, CPN is based on the application of the following modelling concepts [16]: time, hierarchy, and inscription language.

CPN allows the specification of timing properties. This is done by the addition of integer timestamps to individual tokens. Timing annotations in arcs and transitions are used to specify delays usually representing the time an action is modeled to take. The firing of transitions is still an atomic event, but the calculation of enabling for a given transition depends on the timestamps in the tokens it needs to consume through possible input arcs. Intuitively, the time stamp in a token can be seen as a specification of the model time at which the token is available for consumption from a place.

CPN models may be structured hierarchically as a collection of connected modules. A module is itself a CPN model. Structuring is performed through two mechanisms: *fusion places* or *substitution transitions*. A fusion place is a set containing multiple places that may be found in different modules. This allows interaction to traverse boundaries of the modules in the model. A substitution transition is a special transition found in a CPN module that represents an instance of another CPN module. This allows reuse of the specifications of CPN modules throughout the model. The substitution transition is connected to the module in which it is found through an interface which must be common to all instances of the module it represents. This interface is a collection of places. Substitution transitions allow the modeler to work at varying levels of abstraction.

A CPN model is typically annotated by inscriptions in its syntactical components. The inscriptions are used, among other things, to select and manipulate data, and to define functions and data types. In the de facto implementation of CPN, the inscription language is the functional language CPN ML, a close derivative of Standard ML [29].

4.2 An Example of a CPN Model

Figure 6 shows an example of a CPN model. In this case, the model represents a device for measuring and displaying the temperature in a room. This should be done when a user pushes a specific button represented by the *Push Button 1* transition. If this transition fires, a token identifying the button is placed in the *Button Events* place. From this place, button events are consumed by the *Start Measuring* transition, but only if they represent the pushing of button 1 (matching the guard $[b = 1]$). The detection of such an event causes the device to start measuring the room temperature through some sensor. The room temperature is represented by the value of the token in the *Room Temperature* place. This value is continuously modified by the physics of the environment (e.g., the sun) represented very abstractly by the *Modify Temperature* transition. When the measuring is done, the resulting measurement is placed as a token value in the *Measured Temperature* place from where it is ultimately consumed by the *Finish Measuring* transition. The latter causes the measurement to be shown in the display represented by the value of a token found in the *Display* place.

In this example, the interface between the controller and its environment consists in three shared phenomena: the buttons, the room temperature, and the display. We have represented the buttons as a shared event phenomenon controlled by the environment and the temperature is a shared state phenomenon also controlled by the environment. The display is modeled as a shared state phenomenon controlled by the controller. The measured temperature is an example of a machine phenomenon, which is not shared with the environment although it derives from the room temperature phenomenon. This is an important detail since it exemplifies the distinction between the real-world phenomenon (the room temperature) and a machine representation

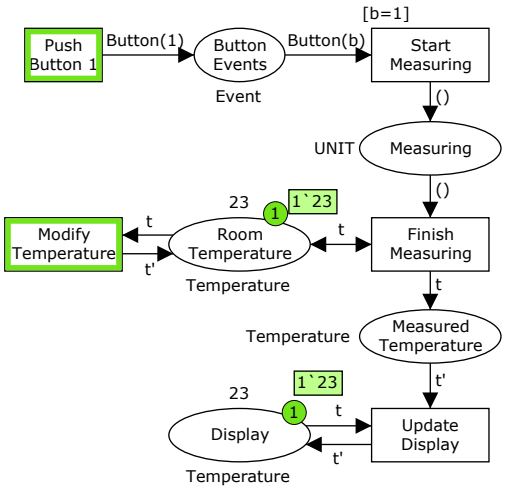

Fig. 6: An example of a CPN model

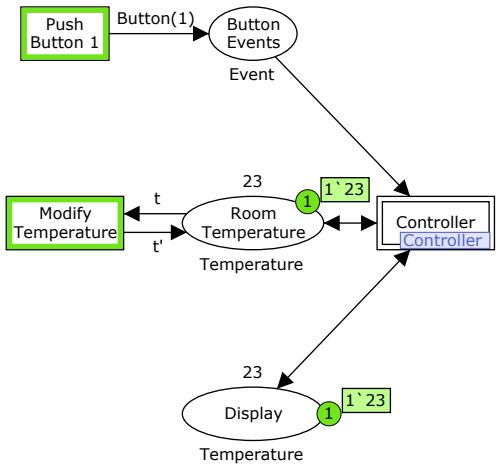

Fig. 7: The controller logic replaced by a substitution transition

thereof (the measured temperature). If the temperature sensor was somehow faulty, these two phenomena would not necessarily be as tightly related as they are in this case. The distinction allows us to take such reliability properties into concern. This approach to representing shared phenomena and thereby distinguishing the environment from the system complies with formalised guidelines that we have defined in earlier work [38] based on the reference model of Gunter et al. [10].

In the examples of CPN shown here, the places carry annotations specifying the data types of tokens that may exist in the places (e.g., *Temperature* and *Event*). These data types are specified in CPN ML as a part of the specification of the model. Also, the two temperature places carry an inscription that specifies the starting temperature (23) as an *initial marking*. This means that before the first step of execution, each of these two places holds a single token of the *Temperature* type with the value of 23.

The principle of hierarchy by substitution transitions is shown in Figure 7, where the control logic of the device has been replaced by a single *Controller* transition. This is a substitution transition - i.e. an instance of the *Controller* module shown in Figure 8.

In the *Controller* module, some places have now been equipped with special labels (e.g., *I/O*) specifying that these places form the interface through which the module may interact with external parts of the model. Each place in the interface is matched to a place in each module where the module in question is instantiated through a substitution transition.

In this way, the level of abstraction is raised in Figure 7, while the behaviour is maintained by placing the lower level details into the module in Figure 8.

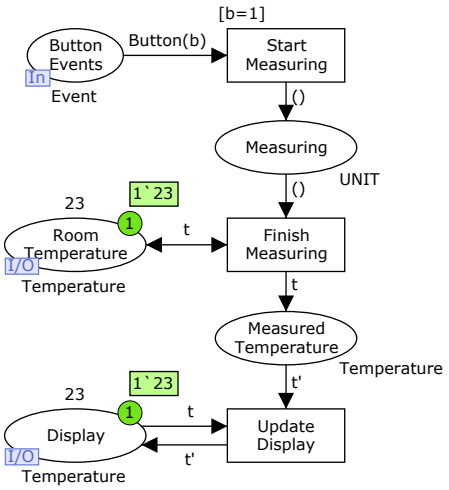

Fig. 8: The contents of the *Controller* module

Figure 9 shows an example of how information about timing properties has been added to the module first shown in Figure 7. First of all, a new data type has been declared in order to add time stamps to the tokens holding the current room temperature in the *Room Temperature* place, i.e., *TemperatureTimed* instead of *Temperature*. In Figure 9, some steps have been executed and the resulting marking is shown. This allows us to see the timestamp in the token found in the *Room Temperature* place: the room temperature is 17 degrees at 5651 time units. We can also see how timing information has been added to the *Modify Temperature*: the $@+discrete(20,45)$ annotation of this transition specifies that the timestamps of tokens produced by firing the transition are increased with an integer value in the interval 20 to 45 time units, picked randomly based on a uniform distribution function. This basically constraints the enabling of the *Modify Tem-*

perature transition allowing us to make a coarse abstraction of the physical properties related to physics affecting the room temperature. The representation of such properties could, of course, be much more detailed and accurate if needed.

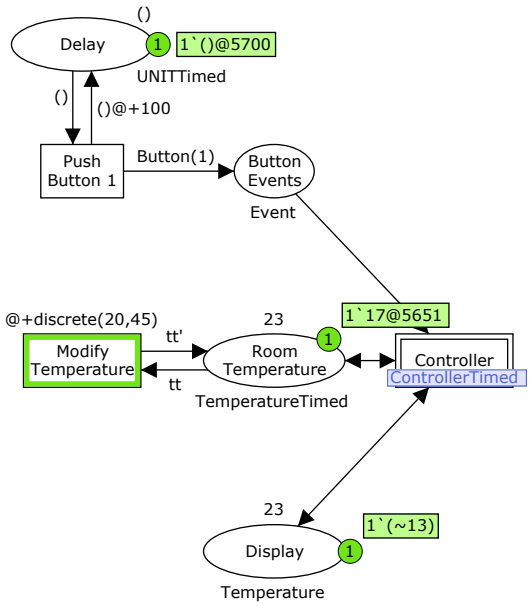

Fig. 9: Adding timing information to the CPN module

Timing information is also added to cause the modelling of button pushes to occur with a fixed interval of 100 time units. This is done by the addition of the *Delay* place that holds a single token only used for delaying the enabling of the *Push Button 1* transition.

After the addition of the timing information, the model reflects a different behaviour when executed compared to the behaviour exhibited before adding the timing information. Now, the temperature is modified with a random delay (within some interval) and the button is periodically pushed.

4.3 On the Adequacy of CPN for EUCs

Firstly, the fact that CPN is a dialect of high-level Petri nets makes it suitable for modelling large real-world problems. This is mainly due to the features related to hierarchy and complex data types. High-level Petri nets are sometimes compared with high-level programming languages with elaborated data types, whereas low-level Petri nets are compared with assembly languages. EUCs based on the use of CPN are immediately applicable to large real-world systems like the PHCS.

Secondly, CPN is well-supported by *CPN Tools* [42], which is a computer tool developed at University of Aarhus. CPN Tools is licensed in more than 4,000 copies, and its users include several hundreds companies.

In the third place, Petri nets' general suitability for describing the behaviour of systems with characteristics like concurrency, resource sharing, and synchronisation tend to trigger attendance to important questions that are useful to deal within the requirements engineering process. CPN provides an extensive state concept, which facilitates the representation of properties of the environment. For example, in the EUC for the elevator controller , it is straightforward to express that the current state of the environment is such that elevator cage 1 is idle at floor 1, cage 2 is stationary at floor 4 with its doors open, and there is an outstanding request for downwards movement for floor 9.

The possibility of easily modelling concurrency was also helpful when modelling the real-world environment of the PHCS, which is indeed highly based on concurrent actions. This makes the EUCs useful for answering questions about the interaction between the system and the nurses. Examples of questions (Q), and corresponding answers (A) of this nature that have emerged at workshops at which the PHCS EUC was used by nurses are: (Q1) What happens if two nurses both are close to the medicine cabinet computer? (A1) The computer generates login buttons and patient list buttons for both of them. (Q2) What happens when a nurse carrying a number of medicine trays approaches a bed computer? (A2) In addition to a login button and a patient list button for that nurse, only one medicine plan button is generated — a button for the patient associated with that bed. (Q3) Is it possible for one nurse to acknowledge pouring of medicine for a given patient while another nurse at the same time acknowledges giving of medicine for that same patient? (A3) No, that would require a more fine-grained concurrency control exercised over the patient records.

With pervasive computing, requirements engineering must deal with new issues such as mobility and context-awareness. Both issues are accommodated in a natural way in a CPN model. Objects like users (for example nurses) and things (for example medicine trays) are naturally modelled as CPN tokens, and the various locations of interest can be captured as CPN places. A CPN state as a distribution of tokens on places is a straightforward modelling of a context affecting the appearance of a pervasive system. Mobility in terms of movements of users and things are described by transition occurrences.

As we argue in more detail in [24], CPN is a modelling language that satisfies four of the five criteria, which Selic puts forward as being essential for good modelling languages in [33]. CPN models (1) are *ab-*

stract, (2) are *understandable* (when used as ingredient in an EUC, that is, hidden behind a graphical animation), (3) can be made *accurate*, and (4) can be used for *prediction*. However, there is no evidence that CPN models satisfy Selic's fifth criteria, that models must be *inexpensive*. The cost-effectiveness of using CPN has not been established well — which, by the way, is an issue that CPN shares with many, if not all, formal methods.

5 CPN Models and Sequence Diagrams

In previous work, we have explored different approaches to combining the use of UML 2 Sequence Diagrams with CPN models. We see this combination as a sensible means to closing the gap between the desired informality of tier 1 and the necessary formality of tier 2. We add formality to sequence diagrams by translating them to executable CPN models and thereby defining their execution semantics. The advantage is that a CPN model representation of one or more sequence diagrams is unambiguous, because of the well-defined semantics of the underlying modelling language.

We describe how a CPN model is systematically generated based on a collection of sequence diagrams in [9]. The structure of all sequence diagrams is translated into a composite CPN model representing all possible behaviour expressed by those sequence diagrams. The focus is on the behaviour of the human actors — e.g., the nurses in the PHCS. In any reactive system, the scenarios of behaviour of the system can be seen as an interplay between the system being designed and its physical environment (including possible human actors). In [9], the behaviour of the environment is implicitly modelled by the introduction of *variation points*, which are used to represent scenarios separating into several potential paths of behaviour. For example, a scenario in which a nurse enters the medicine room could have a variation point leading to two potential paths: in one path, the medicine room is empty when the nurse enters, and in the other path another nurse is already in the medicine room. We provide a method for specifying specific scenarios to be simulated (and animated) at adjustable levels of strictness. In the extreme cases, a walk through a scenario is either completely fixed or completely free. In between, it is possible to make fixed choices for some variation points, while allowing the choice to be free in others. Whenever the choice is free at a variation point, a path can either be determined randomly by the simulation tool or by an interacting user through the animation interface of tier 3.

In [7], we introduce explicit modelling of the assumed behaviour of the physical environment of the system being developed. In order to do this in a structured manner, we describe how the environment and the computer system are modelled in a composite model, in which the interface between the two domains is explicitly identified. In [15], Jackson points out the importance of properly identifying this interface, and of describing both the computer-based system and its environment. As an example, it is important to explicitly distinguish actions performed by the nurse from those performed by the system. This is important for many reasons, one being that requirements can only be rightfully expressed about the behaviour of the system and not about the behaviour of the nurse. No requirements can be made about the behaviour of the nurse, but scenarios can be designed to imitate some thought-of behavioural patterns - i.e. assumptions about the behaviour of the environment. We can provide the nurse with a manual on how to perform specific tasks using the PHCS but because she is a human being acting out of free will, she may exhibit spontaneous behaviour that does not comply with the requirements. On the other hand, such behaviour may be covered by our anticipated scenarios of behaviour.

We further elaborate the approach by the specification of a generic sequence diagram interpreter expressed as part of a CPN model in [8]. This permits to experiment with large collections of sequence diagrams by simply changing parameter values in the CPN model. The assumed behaviour of the physical environment entities are still expressed in terms of specialised CPN structure and the explicitly described interface between environment and system is preserved.

6 Related Work

The EUC approach was first published in [19] in 2003 and has since then been refined in a number of papers. EUCs are, obviously, not a fundamentally new idea. For at least 15-20 years, the basic idea that we use in EUCs, that of augmenting traditional use cases or scenarios with notions of execution, formality, and animation has been well-established in the software industry. A usual prototype based on an informal sketch may be seen as an EUC with the formal tier created in a programming language and the animation tier being the graphical user interface of an application. A detailed comparison of EUCs and traditional prototypes is reported in [3].

Execution and animation of requirements through formalisation in various graphical modelling languages have had and are having attention by the research community, but often the systems considered are small, like the simple communication protocol in [28]. In compar-

ison, EUCs based on the use of CPNs, are, as we noted above, scalable to large real-world systems.

In [12], Harel and Marelly also adopt the term *Executable Use Cases*. Their approach aims at specification of reactive systems through an intuitive way to automatically generate executable and formal models from scenarios. In comparison, our EUC approach focuses explicitly on and strongly emphasises the representation of the environment in which the system must function. EUCs are a manual approach, but we see this characteristic as an important benefit, because the interplay between the three tiers of an EUC not only supports, but actually stimulates communication between users and software developers.

Another example of formalisation at an early stage is found in [34], where the authors annotate use cases and thereby allowing automated translation into low-level Petri nets. This is a typical alternative to our approach where the informal nature of tier 1 is explicitly preserved. Another difference is that the resulting models do not explicitly distinguish the behaviour of the system from that of the environment. Consequently, it is difficult, if not impossible, to identify the specifications without including the assumed behaviour of the environment.

While we strive to distinguish the representations of system and environment, Lauesen describes Task Descriptions [27], where one explicitly postpones the decision about if a given action is performed by the computer or a human actor. This gives advantages in some situations, like for example when the computer-based system is partly constructed from off-the shelf components. In our approach, such components are modeled as having assumed behaviour and are therefore considered as given parts of the environment, rather than parts of the computer-based system being developed. It is relevant to investigate how to perform the move from requirements expressed as Task Descriptions to EUCs in a practical manner. Some preliminary work is presented in [22].

7 Conclusions

This paper presents the concept of *Executable Use Case*, which supports the requirements engineering activities by means of a model-based approach. Structurally, an EUC consists of three tiers. All tiers describe the same reality, but use distinct description languages: tier 1 is an informal description, tier 2 is a formal and executable model, and tier 3 is a graphical animation that just uses concepts that are familiar to and understandable for the users of the system under consideration.

In Figure 1, the bar that indicates the scope of an EUC stretches from the *Real world* bubble to the *Requirements and Specifications* bubble. This means that the end product of applying the EUC approach is a specification, when EUC usage is taken as close to an implementation as possible. The bar that indicates the scope of MDD stretches from the *Requirements and Specifications* bubble to the *Implementation* bubble. In the point where the two bars meet, in the *Requirements and Specifications* bubble, it is indicated that EUCs and MDD can have an overlap. The size and position of the overlap may be subject to discussion, because it is of course possible to pay proper attention to requirements, that is user-level requirements as we have discussed in this paper, in an approach based on MDD. However, if use cases in the sense of specific interactions between external actors and a computer-based system is the first thing that is produced, the MDD approach does not start with describing requirements —problems to be solved— but with making specifications —solutions to be made.

Providing a tighter and stronger connection between EUCs and MDD requires more research in finding good ways to structure the formal tier of an EUC, such that it clearly discriminates between the environment and the computer-based system and such that the part of the model that describes the computer-based system can easily be turned into an implementation. Some work on this topic is described in [1], which (1) presents a CPN model that is used to express user requirements, and then (2) explains how this CPN model is transformed into a design-level CPN model, for describing the behaviour of the software to be developed, and in this way constitutes a specification. Another line of research is to investigate the adoption of other formal modelling languages at tier 2 of EUCs. However, as discussed in Section 4, we believe that CPN models are a proper alternative and present important advantages.

References

1. J.P. Barros and J.B. Jørgensen. A Case Study on Coloured Petri Nets in Object-oriented Analysis and Design. *Nordic Journal of Computing*, 12(3):229–250, 2005.
2. L. Bernardinello and F. de Cindio. A Survey of Basic Net Models and Modular Net Classes. In *Advances in Petri Nets 1992, The DEMON Project*, pages 304–351. Springer, 1992.
3. C. Bossen and J.B. Jørgensen. Context-descriptive Prototypes and Their Application to Medicine Administration. In *5th Conf. on Designing Interactive Systems (DIS 2004)*, pages 297–306, 2004. DOI 10.1145/1013115.1013157.
4. H.B. Christensen and J.E. Bardram. Supporting Human Activities – Exploring Activity-Centered Computing. In *4th Int. Conf. on Ubiquitouos Computing (UbiComp 2002)*, volume 2498 of *LNCS*, pages 107–116. Springer, 2002.

5. S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In *14th Int. Conf. on Application and Theory of Petri Nets (ICATPN 1993)*, volume 691 of *LNCS*, pages 186–205. Springer, 1993.
6. A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2000.
7. J.M. Fernandes, J.B. Jørgensen, and S. Tjell. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. In *14th Asia-Pacific Software Engineering Conf. (APSEC 2007)*, pages 294–301, 2007. DOI 10.1109/APSEC.2007.81.
8. J.M. Fernandes, S. Tjell, and J.B. Jørgensen. Requirements Engineering for Reactive Systems with Coloured Petri Nets: the Gas Pump Controller Example. In *8th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2007)*, 2007.
9. J.M. Fernandes, S. Tjell, J.B. Jørgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *6th Int. Workshop on Scenarios and State Machines (SCESM 2007)*, 2007. DOI 10.1109/SCESM.2007.1.
10. C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, 2000. DOI 10.1109/52.896248.
11. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.
12. D. Harel and R. Marelly. Specifying and Executing Behavioural Requirements: The Play-in/Play-out Approach. *Software and Systems Modeling*, 2(2):82–107, 2003.
13. M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.
14. M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2001.
15. M. Jackson. Some Basic Tenets of Description. *Software and System Modeling*, 1(1):5–9, 2002.
16. K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer, 1992.
17. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Software Tools for Technology Transfer*, 9(3–4):213–254, 2007. DOI 10.1007/s10009-007-0038-x.
18. J.B. Jørgensen. Addressing Problem Frame Concerns via Coloured Petri Nets and Graphical Animation. In *2nd Int. Workshop on Advances and Applications of Problem Frames (IWAAPF 2006)*, pages 49–57, 2006.
19. J.B. Jørgensen and C. Bossen. Requirements Engineering for a Pervasive Health Care System. In *Proc. 11th IEEE Int. Conf. on Requirements Engineering (RE 2003)*, pages 55–64, Monterey Bay, California, 2003. IEEE. DOI 10.1109/ICRE.2003.1232737.
20. J.B. Jørgensen and C. Bossen. Executable Use Cases as Links Between Application Domain Requirements and Machine Specifications. In *3rd Int. Workshop on Scenarios and State Machines (SCESM 2004)*, pages 8–13, 2004.
21. J.B. Jørgensen and C. Bossen. Executable Use Cases: Requirements for a Pervasive Health Care System. *IEEE Software*, 21(2):34–41, 2004. DOI 10.1109/MS.2004.1270759.
22. J.B. Jørgensen, K.B. Lassen, and W. M. P. van der Aalst. From Task Descriptions via Colored Petri Nets Towards an Implementation of a New Electronic Patient Record Workflow System. *Software Tools for Technology Transfer*, 10(1):15–28, 2007. DOI 10.1007/s10009-007-0054-x.
23. J.B. Jørgensen. Executable Use Cases: a Supplement to Model-Driven Development? In *4th Int. Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 8–15, 2007.
24. J.B. Jørgensen. Coloured Petri Nets and Graphical Animation: a Proposal for a Means to Address Problem Frame Concerns. *Expert Systems*, 25(1):54–73, 2008. DOI 10.1111/j.1468-0394.2008.00454.x.
25. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Software Tools for Technology Transfer*, 2(2):98–132, 1998.
26. C. Larman. *Applying UML and Patterns — An Intro to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, 2005.
27. S. Lauesen. Task Descriptions as Functional Requirements. *IEEE Software*, 20(2):58–65, 2003.
28. J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behaviour Models. In *22nd Int. Conf. on Software Engineering (ICSE 2000)*, pages 499–508, 2000. DOI 10.1109/ICSE.2000.870440.
29. R. Milner, M. Tofte, R. Harper, and D. Macqueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.
30. T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
31. C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.
32. W. Reisig. Place/Transition Systems. In *Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I*, pages 117–141. Springer, 1987.
33. B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. DOI 10.1109/MS.2003.1231146.
34. J.R. Silva and E.A. Santos. Applying Petri Nets to Requirements Validation. In *17th Int. Congress of Mechanical Engineering (COBEM 2003)*, volume 1 of *ABCM Symposium Series in Mechatronics*, 2003.
35. I. Sommerville. *Software Engineering.* Addison Wesley, 2007.
36. E.A. Strunk, C.A. Furia, M. Rossi, J.C. Knight, and D. Mandrioli. The Engineering Roles of Requirements and Specification. Technical report, CS-2006-21, Dept. Computer Science, University of Virginia. Also: Technical Report 2006.61, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 2006.
37. P.S. Thiagarajan. Elementary net systems. In *Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I*, pages 26–59. Springer, 1987.
38. S. Tjell. Distinguishing Environment and System in Coloured Petri Net Models of Reactive Systems. In *2nd IEEE Int. Symposium on Industrial Embedded Systems (SIES 2007)*, pages 242–249, 2007. DOI 10.1109/SIES.2007.4297341.
39. R. J. Wieringa. *Requirements Engineering: Frameworks for Understanding.* John Wiley & Sons, 1996.
40. R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML.* Morgan Kaufmann, 2003.
41. P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997. DOI 10.1145/237432.237434.
42. CPN Tools. www.daimi.au.dk/CPNTools.
43. MDA Resource Page. www.omg.org/mda.
44. UML Resource Page. www.uml.org.