Executable Use Cases: a Supplement to Model-Driven Development?

Jens Bæk Jørgensen
Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
E-mail: jbj@daimi.au.dk

Abstract

Executable Use Cases (EUCs) is a model-based approach to requirements engineering. In the introduction to this paper, we briefly discuss how EUCs may be used as a supplement to Model-Driven Development (MDD). Then we present the EUC approach in more detail. An EUC can describe and link user-level requirements and more technical software specifications. In MDD, user-level requirements are not always explicitly described; it is sufficient for MDD that a specification, or platform-independent model, of the software that we are going to develop is provided. Therefore, a combination of EUCs and MDD may have potential to cover the full software engineering path from user-level requirements via specifications to implementations of running computer systems.

Topics: Requirements engineering; requirements and specifications; specification of platform-independent models; Model-Driven Development; Coloured Petri Nets (CPN).

1 Introduction

If we seek support for the full software engineering path from user-level requirements, often based on observations of the real world and informal descriptions, via specifications to implementations of running computer systems, Model-Driven Development (MDD) [19] can offer significant help. MDD focuses on automatic translation of models of software into running implementations on various execution platforms. In this way, MDD is solution-oriented. On the other hand, MDD does not always emphasize the requirements engineering needed to produce the necessary specifications. For example, OMG’s Model-Driven Architecture (MDA) [23] does not provide much support that encourages software developers to pay enough attention to properly identifying and describing the problems that the software must solve.

With this observation as motivation, we will propose Executable Use Cases (EUCs) [13], which is a model-based approach to requirements engineering, as a means that can be used together with MDD. Figure 1 illustrates the relationship between EUCs and MDD.

In Figure 1, the terms requirements and specifications are used in accordance with the terminology of Jackson [7, 8, 21]. 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. A specification is a description of an interaction between the environment and the computer system, for example when the driver steps on the brake pedal, a brake control computer system that we are going to develop will receive a stimulus and, in response, send a signal to the motor to reduce its speed.

A requirement belongs to the users’ world, and does not need to mention the computer system and software in consideration. A specification is often quite technical and a matter for experts rather than average users.

In any software development project, it is essential to pay proper attention to both requirements and specifications. Users must be involved in discussing, eliciting, prioritizing, etc. requirements. Software developers need specifications to have operational starting points for more detailed design and eventually implementation. However, often the requirements are not explicitly formulated — they just exist in the environment without being caught and written down or represented explicitly otherwise. This is in contrast to specifications, which are usually produced in plan-driven approaches to software engineering. An example of a popular means to write specifications is use cases, in the style of UML diagrams [16, 24] or in textual form [4].

One important use of requirements and specifications is to give adequacy arguments for the software
to be developed. For example, if our job is to develop the brake controller we just discussed, we want to argue that if the specification is satisfied (the brake pedal produces a stimuli to the controller, which sends a signal to the motor,...), 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. We cannot influence or change how brake pedals and car motors work — they are given — but it is essential to know their properties, because our controller must interface with them. And it is the composite system consisting of our controller plus the brakes, motors, etc. that must produce the desired effect in the environment.

With EUCs, we provide a means to describe and link requirements and specifications. In particular, EUCs can be used to give adequacy arguments. We will present EUCs this paper, which has the following structure: In Section 2, we introduce EUCs. Section 3 describes two projects in which we have used EUCs. In Section 4, we discuss the formal modelling language of Coloured Petri Nets (CPN) [9, 15] that has been a key ingredient in the EUC projects we have carried out so far. In Section 5, we draw some conclusions, including returning to the discussion about the relationship between EUCs and MDD; we also briefly consider related and future work.

2 Executable Use Cases (EUCs)

An Executable Use Case (EUC) [13] supports description, validation, and elicitation of requirements and specifications. An EUC can represent desired behaviour of the environment (requirements), desired behaviour of the computer system (specifications), and assumed behaviour of external entities in the environment (often needed in adequacy arguments) in the same description.

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 system that happens at the interface of the computer 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. It is this property that enables an EUC to represent both requirements and specifications.

The name Executable Use Cases was chosen to make it easy to quickly and roughly explain the main idea of 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 2, an EUC consists of three tiers.

![Figure 1. Relationship between Executable Use Cases and Model-Driven Development.](image)

![Figure 2. Executable Use Cases.](image)
The tiers describe the same things, but use different representations: Tier 1 (the informal tier) is an informal description; tier 2 (the formal tier) is a formal, executable model; tier 3 (the animation tier) 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 new computer system.

The three tiers of an EUC should be created and executed in an iterative fashion. The first version of tier 1 is based on domain analysis, and the first version of tiers 2 and 3, respectively, is based on the tier immediately below. Tier 1 represents everyday requirements engineering activities, and is created routinely in many projects, often consolidated in the form of, indeed, traditional use cases. Tier 1 should be the result of collaboration between a broad selection of users, software developers, and possibly other stakeholders in the initial pursuit of engineering the requirements for a new computer system.

EUCs are effective means in requirements engineering in conjunction with validation (getting confirmation that the current description is adequate) and elicitation (generation of new ideas and discovery of new issues that should be dealt with).

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 is, in the same way as validation, supported through execution. When users interact with tier 3, they will 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 prose 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 EUC Projects

In this section, we will present two projects in which we have used EUCs. In line with the focus of MOM-PES, we consider an embedded system, namely an elevator controller, and a pervasive system, namely a new hospital computer system. It should be noted though, that EUCs are applicable to traditional administrative systems as well, see for example [14].

3.1 An Elevator Controller

The elevator controller is a standard text book example; our version has been taken from [20]. The main responsibility of the controller is to control the movement 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 movement of the cages, the controller is responsible for updating a location indicator inside each cage, which displays the current floor of the cage.

We have discussed the elevator controller and its EUC in a number of papers, for example in [10]. 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 shown in Figure 3 for 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, it is triggered that graphical objects like cage icons are moved, their graphical appearance changed, etc. 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;
- **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 will 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 there currently 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 floor buttons, sensors, motors, and doors. In the animation tier, only the consequences of the technical specifications are visible.

A user can push floor and cage button icons in the animation tier. For each push, the user will experience 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 problems like an elevator cage, which does not stop if it comes to a floor for which it has a request, or more complex problems like the scheduling not being done so that efficient use of the elevator cages is ensured.

However, the animation tier 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.

3.2 A Pervasive Health-care System

In contrast to the elevator controller, which in our version above is a fictitious system from a text book, the pervasive health care system (PHCS) [3] is a real system aimed at use at Danish hospitals. The objective of PHCS is to ensure smooth access to and use of hospital computer systems by taking advantage of pervasive computing.

PHCS is context-aware. This means that PHCS 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 by the medicine cabinet and by the patient beds.

Another property of 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 by 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 medicine administration, covering nurses’ pouring and giving of medicine. The EUC described how medicine administration is supposed to be supported by PHCS. The use of EUCs in requirements engineering for PHCS is described in detail in [11, 12].

The animation tier of the EUC, which is shown in Figure 4, 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 nurses interaction with them.

Examples of PHCS requirements 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.

Note that these requirements are genuinely technology independent properties 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 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, in its own right, is an important argument for explicitly distinguishing between requirements and specifications.

Examples of PHCS specifications 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.

Again, 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 will experience 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 in just two clicks; first on the patient list button, and then on the name of the patient of concern. Thus, if a computer system is constructed that meets (S1), and the computer system has a reasonable performance, (R1) will be satisfied. Similarly, the EUC links requirement (R2) and specification (S2); and requirement (R3) and specification (S3), respectively.

4 Coloured Petri Nets 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 [5], UML state machines or activity diagrams [24], or Petri nets. These languages differ in a number of ways. In particular, they have different degrees of formality and rigidity.

In our use of EUCs so far, and in particular in the two projects described in Section 3, we have used the modelling language Coloured Petri Nets (CPN) [9, 15] as tier 2 language. We have chosen CPN because we have experience with this language and its tool support and because CPN is appropriate for EUCs, as we will argue below.

First of all, CPN is one dialect of high-level Petri nets, a class of Petri nets which makes modelling of large systems tractable. 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 use of CPN are immediately applicable to large real-world systems like PHCS.

Secondly, CPN is well supported by computer tools. The tool support is provided by CPN Tools [22], which
is developed at University of Aarhus as a successor to the commercially developed tool Design/CPN. CPN Tools is licensed in more than 4,000 copies, and its users include several hundreds companies.

In the third place, CPN provides an extensive state concept, which facilitates the representation of properties of the environment. For example, in the elevator controller EUC, 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.

In the fourth 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 it is useful to deal with in requirements engineering. Examples of questions (Qs), and corresponding answers (As) 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 effecting 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 [10], CPN is a modelling language that satisfies four of the five criteria, which Selic puts forward as being essential for good modelling languages in [19]. CPN models are (1) abstract, (2) 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. Although many good reasons to use formal methods can be given, for example that they facilitate finding critical errors and flaws early rather than late, the survey [18] of the state-of-the-practice of requirements engineering in the software industry finds that formal models are only rarely used.

5 Conclusions

The EUC approach we have presented in this paper is, 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 [2].

Execution and animation of requirements through formalisation in various graphical modelling languages have had and are having some attention, but often the systems considered are small, for example the simple communication protocol of [17]. In comparison, EUCs based on use of CPN, are, as we noted above, scalable to large real-world systems.

The paper [6] also uses the term executable use cases. It aims at specification of reactive systems through an ingenious, intuitive way to automatic generation of executable, formal models from scenarios. A calculator is the running example. In comparison, our EUC approach focuses explicitly on and strongly emphasises representation of the environment in which a new system must function. Moreover, EUCs are a more mundane, manual approach, but we see the manual aspect as an asset, because the interplay between the three tiers of an EUC not only supports, but actually spurs communication between users and software developers.

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 application of 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. 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 system is the first thing that is produced, the MDD-based approach does not start with describing requirements — problems to be solved — but with making specifications — solutions to be made.

To provide a tighter connection between EUCs and MDD demands 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 system and such that the part of the model that describes the computer system can easily be turned into an implementation. Preliminary work on this topic is described in [1], where we first present a CPN model that is used to express user requirements and then describe how this CPN model is transformed into a design-level CPN model, which describes the behaviour of the software to be developed, and in this way constitutes a specification. Another line of research is to investigate use of other formal modelling languages at tier 2 of EUCs. However, as discussed in Section 4, we believe that use of CPN models in EUCs in many cases have important advantages, which often outweigh their drawbacks.

References


