Unified Use Case Statecharts: Case Studies

Davor Svetinovic[1] Daniel M. Berry[2] Nancy A. Day[2]
Michael W. Godfrey[2]

June 4, 2007

Abstract

This paper presents the results of case studies evaluating a method of unifying use cases (UCs) to derive a unified statechart model of the behavior of the domain of a proposed computer-based system. An evaluation of the unification method, the obtained statechart model of the domain, the method's and model's feedback on the UCs themselves, and how the method is used in requirements engineering practice was carried out by examining 58 software requirements specifications produced by 189 upper-year undergraduate and graduate students. The results of these studies independently confirm some of the benefits of building a unified SC mentioned in the works of Glinz; Whittle and Schumann; and Harel, Kugler, and Pnueli.

1 Introduction

Analyzing and specifying the behavior of a computer-based system's (CBS's) domain is a very hard requirements engineering (RE) task. Teaching people how to perform this task is even harder [14].

The goal of an RE effort is to elicit and analyze requirements, and eventually to specify in a Software Requirements Specification (SRS) document CBS's desired behavior and properties, i.e., the CBS's *requirements*. The portion of the

[1]Institute of Computer Technology, Vienna University of Technology, A-1040 Vienna, Austria.
[2]David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

real world that a CBS is supposed to automate is the CBS's *domain*. During RE analysis for a CBS, the analysts typically develop the *behavioral domain model*, which is a model of the behavior of the CBS's domain. In use-case-driven requirements analysis methods [e.g., 15], the first task analysts perform in modeling the behavior of the CBS being built is to write *use cases* (UCs) that describe the CBS's intended behavior. A UC of a CBS is one particular way some user of the CBS uses the CBS to achieve stakeholders' goals. Domain experts and analysts together typically capture UCs during and after requirements elicitation from many stakeholders, each with a different perspective. The description of a UC is typically given at the shared-interface level, showing the CBS as a monolithic black box. From these UCs, the analysts begin to model the entire CBS's domain. A popular notation for modeling behavior is *statecharts* [9], and among the artifacts that can be included in the SRS for a CBS, are a *UC statechart*, a statechart representation of each UC of the CBS; and a *unified UC statechart*, which is a statechart representation of the CBS's domain. In object-oriented analysis (OOA), the analysts break down and describes the entire CBS's domain in terms of objects that are used later as a main source of objects for object-oriented design (OOD). *Conceptual analysis* is this whole process of discovering and specifying concepts from a domain.

Each of the authors has been involved in teaching a course titled "Software Requirements Specifications" (CS445)[3] at the University of Waterloo for some periods during the last six years. The term-long group project for this course is to determine the requirements and to write an SRS for a large VoIP system and its information-management system (IMS). The specification of this CBS involves using a use-case-driven approach that results in an SRS with

1. formal finite-state modeling in SDL [1] for the real-time VoIP system components and

2. the notations of UML [19] for the IMS components.

In addition, students are responsible for modeling user interfaces of the IMS and for the overall management of the requirements specification process. The average size of the resulting SRS document for the whole CBS is about 120 pages, with actual sizes ranging from 80 to 250 pages. The estimated effort required for a proper specification of this CBS is approximately 400 person hours.

During the last six years, we have observed the production of and have evaluated the SRSs, including UC specifications, produced by 910 students, working

[3]`http://se.uwaterloo.ca/~dberry/cs445w06`

in 3- or 4-person teams. We have observed that the typical result of this initial UC capture is a set of UCs with missing functionality, unrelated functionality across multiple abstraction levels, inconsistent amounts of detail in the form of over and under specification, and problems arising from the difficulty of abstracting from multiple UCs to the big picture of the domain. These observations are consistent with those of other authors [e.g., 17]. In short, the quality of the UCs is not good. Specifying consistent, high quality UCs is hard.

Specifying high quality UCs is necessary because of their central role in UC-driven requirements analysis methods, such as OOA. In these methods, UCs drive subsequent analysis, design, and coding. Any problem with the UCs propagates through the rest of the development process and infects the artifacts that are produced. Therefore, it is essential to expose problems with UCs as early as possible.

Our observations let to some preliminary studies of the nature of the difficulty [23]. Svetinovic began his Ph.D. research to find a way to teach the students to do a better job in their projects. We explored the literature on UC-driven analysis methods [e.g., 4, 8, 15]. We examined different variations of these methods on some CBS domain modeling problems. We experimented with advice to give to the students about these methods. With the students' help and feedback, we slowly iterated to a method that our students were able to apply and with which the quality of the resulting SRSs was noticeably improved. The method, which builds on using statecharts to model UCs and then unifying the UC statecharts into a unified UC statechart [6, 26, 11], is called UCUM (Use Case Unification Method). One of us, Berry, taught a graduate course titled "Advanced Topics in Requirements Engineering" (CS846)[4] once at the University of Waterloo. One of the goals of this course was to explore the impact of method modifications in a more controlled environment than possible in the CS445 course. The students were asked to apply the methods of CS445 on a smaller problem, the controller for a two-elevator system in a low-rise building.

This paper describes the thinking that led to unified UC statecharts and to UCUM and includes a detailed description of the authors' observations of a group of students working through the process of building a unified UC statechart. The usefulness of unified UC statecharts and the effectiveness of UCUM were validated through evaluation of 58 SRSs specified by 189 upper-year software engineering, electrical and computer engineering, and computer science undergraduate and graduate students. Out of these 58 SRSs, 46 were produced by 3- or 4-person teams and 12 SRSs were produced by single-person teams. Each of

[4]`http://se.uwaterloo.ca/~dberry/ATRE/ElevatorSRSs/`

these 189 students had none to several years of software development experience.

Section 2 reviews the problems with UC-driven requirements analysis methods as we experienced them. Section 3 explains the research method and its limitations. Section 4 describes unified UC statecharts, the main artifact produced in UCUM and UCUM itself. Section 5 presents a very detailed first case study of UCUM use. Section 6 evaluates UCUM as a method by examining the first and two other case studies of UCUM use. Section 7 describes related work, and Section 8 concludes the paper.

2 The Problem

A traditional *non*-UC-driven RE method focuses on eliciting and specifying functional, data, and non-functional requirements as distinct entities, without really considering their context. Such a method results often in a SRS that is difficult for both users and CBS's designers to understand.

The failure to explicate the connections among the different kinds of requirements makes it difficult to determine if the SRS is complete, consistent, and correct. UCs [e.g., 12, 2] have helped solve some of these problems, at least for functional requirements. The ability to integrate and present functional requirements from the users' perspectives in UCs has made UCs particularly useful for users. Because UCs present functional requirements as observed by a user, it is easier to identify missing functions, and makes it possible to write a more consistent and complete SRS that is understood by both the user and the analyst [e.g., 2].

In the typical UC-driven requirements analysis method, UC discovery is followed by drawing sequence diagrams for the UCs [e.g., 15] and then creating a conceptual model and a behavioral model per concept. This kind of approach had been taught to the CS445 students for several years. A less common alternative method is to follow UC discovery by drawing UC statecharts [4, 15, 8]. Nevertheless, in each method, a UC is the artifact at the widest *scope*. *Scope* refers to the number of functional requirements captured and specified using an artifact. A sequence diagram for a UC, a UC statechart, or any other description of a UC is at a scope equal to or less than that of the UC, i.e., it captures at most the same number of functional requirements and relationships among them as does the UC. To arrive at the big picture behavioral domain model, it was necessary to proceed in the opposite direction, i.e., to widen the scope captured in an artifact.

Following this direction, we realized that maybe some *new artifact* based on the UCs would allow analysts to produce *even* more complete, consistent, and correct SRSs, and behavioral domain models in particular. Perhaps, the same way that UCs help put functional requirements into context, this new artifact based on the UCs would help an analyst to achieve the following five goals:

- detecting and fixing missing functionality,
- detecting and fixing functionality across multiple abstraction levels,
- detecting and fixing inconsistent amounts of detail, i.e., over and under specification,
- discovering relationships, e.g., concurrency among UCs and functional requirements and concepts, and
- finding a *big picture* behavioral domain model.

We realized that instead of decomposing the UCs, as suggested in many UC-driven requirements analysis methods, it might be better to unify the UCs into a behavioral domain model using statecharts, as suggested by others [6, 26, 11].

Therefore, our hypothesis is that unifying the UCs into a behavioral domain model using statecharts helps an analyst to achieve the five goals discussed in the preceding paragraph.

3 Research Method

The idea of unifying UCs into a unified UC statechart is due to Glinz; Whittle and Schumann; and Harel, Kugler, and Pnueli [6, 26, 11]. The students did not follow any of these authors' proposed methods per se. Instead, the students used the underlying UC integration idea and basic integration steps common to all these methods to manually unify UCs into a unified UC statechart. Rather than having students follow a strict method, the idea was to have students tackle building unified UC statecharts as an engineering problem that they had to solve. The first author of this paper used a small domain as an exercise during the tutorials to have students work from scratch on applying and refining a simple manual method, i.e., UCUM, for unifying use cases into a unified UC statechart. This exercise is the Turnstile case study described in Section 4, which describes UCUM and discusses the results of this initial exercise of specifying a behavioral domain model using statecharts.

UCUM was presented and refined during three tutorial sessions with almost 140 students in attendance over the three sessions. UCUM was tailored through practical work on a concrete domain. The way UCUM was presented allowed a student to observe the reasoning of other students and evaluate pitfalls that would help her complete her main course project. Only after specifying UCUM and only after completing all case studies, we compared the students' results with those of the work that inspired UCUM. This comparison is presented as part of the description of related work in Section 7.

3.1 Threats to Validity

The contribution of this paper is a retrospective, after-the-fact analysis of the requirements engineering exercises performed during the courses. No experimental controls were applied during the courses, and there were no restrictions on the students' behavior beyond the normal restrictions applied during in-class exercises and in long-term course projects, e.g., standard measures to prevent academic dishonesty. In particular, the course staff

1. did not require the strict use of the method presented in tutorials by the students, beyond the basic core UCUM,

2. did not require any particular group organization or division of work, and

3. did not limit the size of the CBS, of the CBS's domain, or of any artifact produced.

Finally, each analyzed behavioral domain model was just one part of a complete SRS produced in the exercises. With respect to the first non-exercised control, not only was the exclusive use of UCUM not enforced, but students were in fact encouraged to extend and adapt UCUM as they progressed with their projects and obtained feedback on UCUM itself. Clearly, there is too much variability for these case studies to be considered controlled experiments. Thus, the threats to validity of these case studies are exactly the same as in any other uncontrolled software engineering study.

Nevertheless, the large number of subjects and the high consistency of the results of the case studies in spite of all the variability provides strong support for accepting the finding of the case studies.

The more important question is whether the results presented in this paper can be generalized to apply to practicing professional software engineers. Given that

1. each student subject of the case studies was an upper year undergraduate or a graduate student; the typical undergraduate student subject had from one to two years of industrial software engineering experience through the University of Waterloo's co-operative education program; and an occasional graduate student had worked as a software engineer prior to coming to graduate school,
2. each of statechart modeling and UC modeling is applicable to a broad class of problems,
3. each of statechart modeling and UC modeling is widely used in industry, and
4. each of statechart modeling and UC modeling is widely taught in universities,

the results present here could be generalized at least to apply to a recently graduated software engineer who is joining a new or already existing software development project in which statechart modeling or UC modeling is applied.

4 Unified UC Statecharts

UCUM is based on a simple idea inspired by observing practice: an effective way to unify a complete set of UCs into a behavioral domain model for the CBS is to perform the unification in the statechart notation. That is, if each UC in the set can be described with a UC statechart [e.g., 4, 8], then it should be possible to merge these UC statecharts into a unified UC statechart that serves as a high quality behavioral domain model[5] of the CBS [6, 26, 11].

The method depends on the analysts' having specified the UCs' behaviors in UC statecharts. However, after practice, an analyst can learn to proceed directly from UCs to a unified UC statechart without having given UC statecharts for the UCs. Indeed, we found many a student skipping the production of UC statecharts and still producing a good unified UC statechart. Douglass [4] summarizes the advantages of specifying a UC's behavior using a UC statechart:

[5]Note, this model should not be confused with any high-level business model. The difference is in the abstraction and decomposition levels.

Another means by which UC behavior can be captured is via statecharts. These have the advantage of being more formal and rigorous. However, they are beyond the ken of many, if not most, domain experts. Statecharts also have the advantage that they are fully constructive—a single statechart represents the full scope of the UC behavior.

Statecharts have an additional advantage of being able to help an analyst to unify a set of UC statecharts into a single unified UC statechart. Unifying UCs using a single unified UC statechart widens rather than narrows scope in terms of the number of functional requirements taken into consideration, i.e., a single unified UC statechart binds together the requirements from all UCs rather than just requirements from one UC. Widening scope leads to exposing problems that might still exist in the individual UCs in the same way that widening scope during the unification of functional requirements leads to exposing problems that exist in the individual functional requirements. This integration method produces a model of increased complexity since it captures a larger number of functional requirements and their relationships than without the method. Producing this model and managing its larger complexity facilitates detecting missing requirements and inconsistencies.

The rest of this section discusses the semantics of unified UC statechart models and then describes the process of UCUM. Note that we defined the semantics of unified UC statechart models only *after* all the case studies were finished. We *started* with a simple semantics in which a state is either

- any configuration of variable values, or
- an activity of interest,

but this definition was insufficient for the specification of the unified UC statecharts for even a small CBS due to the additional information that needed to be captured as part of unified UC statecharts.

4.1 Unified UC Statechart Model Semantics

A statechart is a *higraph*, a general kind of diagramming object based on graphs and sets [10], that can be used to model different aspects of a software system. Thus the first, and most important, step in using statecharts is to clearly state *what* is being modeled. An explicit agreement is needed on what a state represents.

While there are various definitions of “state” in the literature, Douglass’s definition summarizes a common view: “A state is an ontological condition that persists for a significant period of time.” [4] In practice, a state is used to capture any configuration of the object’s variables or any activity occurring within the system being modeled [e.g., 4]. For the purposes of behavioral domain modeling, the most appropriate semantics for a state in a unified UC statechart is as a *postcondition*, as a reflection of goals, as in goal-driven RE [e.g., 18, 24].

Goal-driven RE focuses on identification of the goals, as a prerequisite for requirements specification. Goal-driven RE focuses on ensuring that the CBS being built actually fulfills the users’ goals. This focus requires shifting away from considering *what* a CBS should do to considering *why* the CBS should do what it does. In other words, the main focus is on *requirements rationale*.

Although goal-driven RE focuses on determining CBS requirements through analysis of users’ *personal* and *business* goals, goal-driven RE has been used to enhance traditional RE methods, among which are UC-driven requirements analysis methods [e.g., 2]. In the case of UCUM, it was natural to start by determining the UCs for a CBS being built by considering the goals for the CBS. The preservation of the goals, and postconditions in particular, as states in the unified UC statechart followed.

Goals capture the *intentions* and the *target conditions* for the entity under analysis. For example, in the case of an elevator system, a goal for an elevator is to deliver passengers to their requested floors. This goal captures both the *intention* of delivering passengers and the *target condition* of arriving at the passengers’ requested floors. This particular goal captures the rationale for an elevator’s *responsibility* for carrying each passenger from a floor to a floor.

In other words, a UC’s goals are achieved through a sequence of activities each of which is described by a functional requirement. Each goal can exist at an abstraction level different from those of other goals. For example, continuing with the elevator CBS example, the decomposition of the goal deliver passengers to their requested floors might include such lower-level goals as move elevator cab, stop elevator cab, pick up a passenger, etc. That is, the higher-level goal of delivering passengers to their requested floors becomes a functional requirement for the lower-level goals in its goal decomposition. Thus, the goal decomposition hierarchy provides traceability among the goals.

Therefore, a state in a unified UC statechart for a CBS’s domain is more general than a traditional state, which is only a configuration of values of CBS variables and which can be very tedious to specify when there are many variables in a CBS. A state in a unified UC statechart can be either

- an activity in the CBS, or
- a goal that captures the target condition of a part of the CBS or of the entire CBS.

In the latter case, the goal represents the *postcondition* that describes the impact that the activity in the previous state or on the incoming transition has on the CBS. Therefore, we note that the semantics of transitions in unified UC statecharts is consistent with that in traditional statecharts, while the semantics of states differs. In addition, in each presented case study, each statechart conforms to UML 1.x or UML 2.0 statechart syntax and semantics [19]. This underlying behavioral domain model semantics is consistent with Glinz's [5, 7].

"Why not use UML activity diagrams [19] instead of statecharts?" was asked many times because of the presence of states representing activities in the unified UC statecharts. There are several reasons:

- The activity diagram notation is harder to use because of its different interpretations; e.g., an activity diagram can be viewed as a statechart, as a Petri net, or as a flowchart [4].
- Laying out and managing a large activity diagram is more difficult, in our experience, than laying out and managing a large statechart.
- By definition, it is harder to show, using *activity nodes*, anything but activities in an activity diagram [4], implying that it is harder to show different abstraction levels in an activity diagram for anything but activities.
- For any interactive system, there can be many external asynchronous and internal synchronous events, in addition to the implicit activity-completion events for which an activity diagram is tailored, and these are all easier to represent using statecharts.

Moreover, we did not find any feature provided by activity diagrams that is not provided by statecharts.

4.2 Process

UCUM is derived from several sources. UCUM is primarily Larman's UC-driven iterative method [15]. The principles of constructing a unified UC statechart are based on Douglass's and Gomaa's principles of UC statechart construction [4, 8].

UCUM emerged from our literature review, practice on our own examples, and the preliminary work with students on specification of a Turnstile CBS, that is described in Section 5. We recommended UCUM to the students for their projects, and encouraged them to modify UCUM based on the experiences gained through their work. The sequential ordering of steps given here is only for exposition purposes. The students were taught both sequential and iterative processes, and each unit, individual or group, was allowed to use whatever it thought would be more effective. We now define UCUM:
For the CBS S to be built:

Step 1: Specify UCs:

- Identify S's main *goals* and *UCs*.
- For each of S's UCs, U, write a clear description of U with indications of U's *actors*; the *data* exchanged in U between S and S's environment; and U's *preconditions*, *postconditions*, and *invariants*.
- Draw a UML *UC diagram* showing all of S's UCs, to emphasize the relationships that exist among the UCs.

Step 2: Group UCs into domain subsystems:

- Group the UCs into domain subsystems according to the UCs' business concerns. This grouping yields the first level of the decomposition of S's domain D into groups of related business concerns, i.e., the first-level domain subsystems of D.
- Show the decomposition of the UC diagram using UML package notation.
- Repeat Step 2 for any domain subsystems of any level of D that can be further decomposed.

Step 3: Draw UML *system sequence diagrams* [15] for the UCs of S, in order to be able to identify D's external interface. In each of these system sequence diagrams, S is considered as a black box. For each UC U, draw U's UML system sequence diagram, in order to be able to identify U's contributions to D's external interface.

Step 4: Specify the unified UC statechart:

- Merge the activities of all UCs of S to build a unified UC statechart for S's domain, D, either (1) directly or (2) by drawing a UC statechart for each UC of S and then merging all these UC statecharts into a single unified UC statechart.
- If any problem is detected in any UC during the building of the unified UC statechart for D, then fix the UC. These problems can include, but are not limited to, abstraction level clashes, missing operations, redundant operations, inconsistent terminology and improper ordering of operations.
- Simplify the unified UC statechart using concurrent and sub-machine states.

To reduce unified UC statechart rework due to activity refinements, we used the rule: If an activity clearly needs no further decomposition then model it as a transition action, a state's internal action, or a state's internal activity; otherwise model it as a sub-machine state.

The next section shows an example of an application of UCUM.

5 Turnstile Case Study

The Turnstile case study (TCS) concerns the collaborative production of three unified UC statecharts during three tutorial sessions of the CS445 and CS846 courses[6]. The three unified UC statecharts were of the same CBS, the Turnstile CBS, described in an example SRS at the course Website[7].

The starting point for building unified UC statecharts was the set of UCs identified in the example SRS. The first unified UC statechart was produced by 12 CS846 students, the second unified UC statechart was produced by about 80 CS445 students, and the third unified UC statechart was produced by about 50 CS445 students.

The primary value of the exercise was observing:

- three different groups of about 140 students altogether thoroughly analyzing a small CBS's domain to produce unified UC statecharts, and

[6]Note, the case study description includes all steps as recommended to the students for their projects and as described in Section 4.2. Some of the steps in this case study are simplified and do not fully conform to the steps of the process either due to the way initial UCs were specified, e.g., there are no specified goals in Step 1, or due to the simplifications made due to the simplicity of the problem itself.

[7]`http://se.uwaterloo.ca/~dberry/cs445w06/turnstilesystem.pdf`

- the feedback the production of these unified UC statecharts had on the UCs in the earlier, Website-published, and supposedly polished UCs[8].

The goal of each session was to help the students learn UCUM by a process of facilitated collaborative self-discovery of the steps necessary to produce a unified UC statechart. While the teaching assistant, who is the first author of this paper, tried his best to let the students go where they wanted, the author[9] did step in to prevent them from going too far astray, and the author did ask some leading questions that helped students notice things that the author could see they were overlooking.

It was valuable also to see the quality of unified UC statecharts produced by undergraduate students who were novices at statechart modeling. The case study showed also the amount of improvement in the quality of modeling that can be expected when many people are attacking a small problem.

The diagrams in the rest of this section are cleaned up from those produced during the third and final tutorial session.

5.1 Step 1

The first step is to *specify UCs*. Figure 1 shows the three main UCs from the original Turnstile SRS[10] and the context diagram with a domain boundary definition, based on UC descriptions. There are two identified actors: Visitor and Operator. Operator is the initiator of the UC2 and UC1, and Visitor is the initiator of the UC3. Turnstile consists of the Turnstile hardware and the control software, and it serves as the ≪system≫ in the context diagram and as System in the UCs[11].

5.2 Step 2

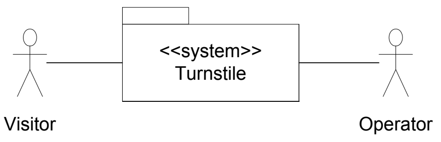
The second step is to *group UCs into domain subsystems* according to their business concerns in order to produce a more refined decomposition of the CBS's domain into domain subsystems. An effective way to document these domain subsystems is with UML packages superimposed on the UC diagrams. One can

[8]`http://se.uwaterloo.ca/~dberry/cs445w06/turnstilesystem.pdf`

[9]The unadorned noun "author" in the rest of Section 5 refers to the first author of this paper who guided these tutorial sessions.

[10]`http://se.uwaterloo.ca/~dberry/cs445w06/turnstilesystem.pdf`

[11]`http://se.uwaterloo.ca/~dberry/cs445w06/turnstilesystem.pdf`



UC1: Turn Off System
1. *Operator* turns off Turnstile
2. *System* resets counters for available entries and visitors
3. *System* locks barrier
4. *System* stops responding to events except “turn on”, shall return payment immediately if it was made at this point

UC2: Turn On System
1. *Operator* turns on Turnstile
2. *System* (again) accepts external events

UC3: Enter Area
1. *Visitor* inserts payment
2. *System* concludes payment can purchase one entry
3. *System* updates the number of available entries
4. *System* unlocks barrier
5. *Visitor* pushes barrier
6. *System* rotates barrier
7. *System* notes visit, increments visitor count
8. *System* locks barrier

Figure 1: Use Cases

further decompose domain subsystems into lower-level domain subsystems that correspond to more refined groupings of UCs into business concerns.

Figure 2 shows the main UC diagram with the CBS’s domain boundary defined. During the tutorials, students attempted to group UCs according to the UCs’ business concerns, but the students were not able to reach consensus. Therefore, the students decided to proceed without the grouping UCs, with the understanding that the grouping could be done later if necessary. The only proposed grouping, suggested during one of the tutorials, was to place UC2 and UC1 into one subsystem and to put UC3 into another. This breakdown made sense because the common business concern of UC2 and UC1 is *managing systems status*, while the business concern of UC3 is *controlling access to restricted area*. Yet, many students did not agree to this grouping since they perceived UC2 and UC1 as supporting UC3 and thus having the same business concern as the UC3. In addition, the students who were in favor of the proposed grouping had difficulty in naming the resulting domain subsystems properly.

Grouping UCs into domain subsystems is neither required nor crucial because this grouping only facilitates further decomposition of the domain into concepts during conceptual analysis, i.e., conceptual analysis can be done even without grouping of UCs into domain subsystems. Therefore, given that time was limited, the students decided to proceed with following steps and come back to this step later if necessary. Also, the small size of the Turnstile CBS made it difficult to group UCs into meaningful domain subsystems, i.e., the decomposition of domain into domain subsystems did not seem essential due to the CBS’s small size. This

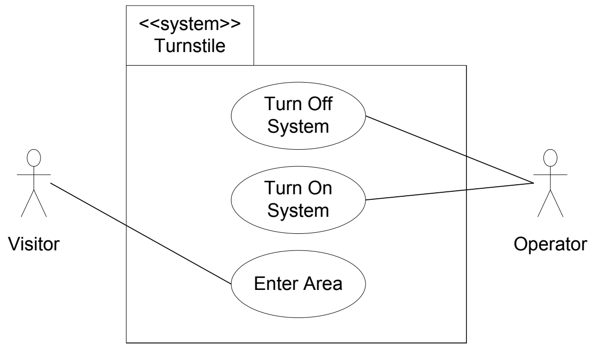


Figure 2: Use-Case Diagram

specification was perfectly clear without the help of the domain subsystems.

5.3 Step 3

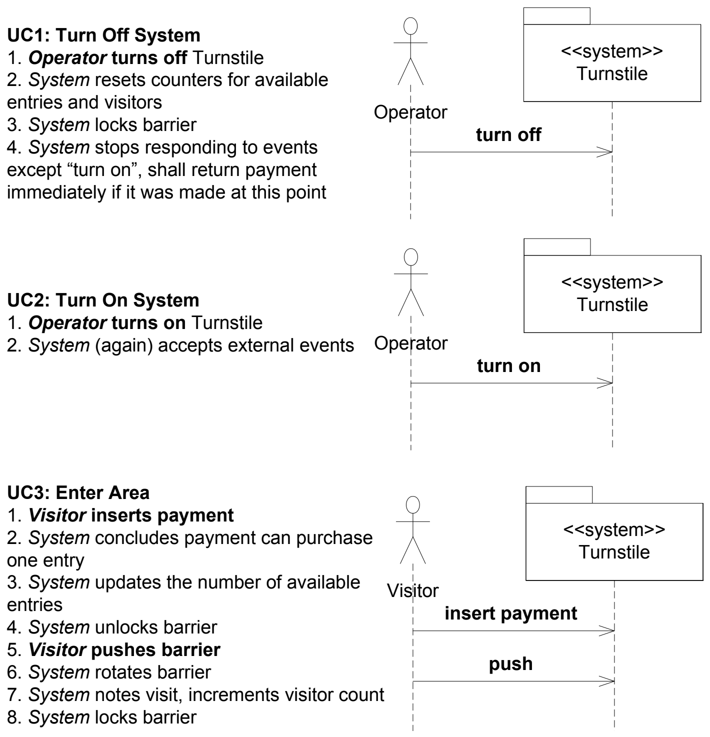
The third step is to *define UC system sequence diagrams*. The goal of this step is to define the domain's external interface. It is important to clearly identify *input* and *output* events for the specification of the unified UC statechart in the next step.

Figure 3 shows the UC system sequence diagrams for the three identified UCs. The students detected only external interfaces that capture *input* to the system. There were several indications of possible *output* from the system, in Operations[12] 2 and 3 of UC1, and Operations 3, 4, 5, 6, 7 and 8 of UC3. These outputs were not included in the diagrams because a majority of the students considered them to be internal communication rather than communication between the domain and domain's environment.

5.4 Step 4

The fourth step is to *specify the unified UC statechart*. This step is the largest and most important in UCUM and is also the most difficult. For each UC, the students had to merge the UC with the unified UC statechart built so far. The students had the option to specify the UC's statechart separately before merging it with unified UC statechart.

[12]A step of a UC is called an "operation" to avoid confusion with a step of the construction carried out in the TCS.



UC1: Turn Off System
1. ***Operator* turns off** Turnstile
2. *System* resets counters for available entries and visitors
3. *System* locks barrier
4. *System* stops responding to events except “turn on”, shall return payment immediately if it was made at this point

UC2: Turn On System
1. ***Operator* turns on** Turnstile
2. *System* (again) accepts external events

UC3: Enter Area
1. ***Visitor* inserts payment**
2. *System* concludes payment can purchase one entry
3. *System* updates the number of available entries
4. *System* unlocks barrier
5. ***Visitor* pushes barrier**
6. *System* rotates barrier
7. *System* notes visit, increments visitor count
8. *System* locks barrier

Figure 3: System Sequence Diagrams

If the students were to detect any problems with the UCs, the system sequence diagrams, or the system boundary definition, then they were supposed to fix them. Most often, these problems were found in the UC currently being currently merged with the unified UC statechart — problems such as inconsistent abstraction levels, missing operations, redundant operations, and improper step order. After each UC was merged, the students were advised to attempt to simplify the unified UC statechart using concurrent and sub-machine states.

Additional advice the students received included:

- Use a *statechart action* to capture an activity only if it is *certain* that there is no need for further decomposition of that activity. Otherwise, the activity should be modeled as a sub-machine state.

- Use a *statechart internal action* or a *statechart internal activity* to capture an activity only if it is *certain* that there is no need for further decomposition of that activity. Otherwise, the activity should be modeled as a sub-machine state.

The purpose of these recommendations was to minimize required rework if an activity were to need further decomposition.

Figures 4 through 21 show the step-by-step construction of the unified UC statechart for the Turnstile. The description of each step discusses all choices for the step, the resulting artifact, and the impact of the step on other artifacts.

Figure 4 shows that students decided to start building the unified UC statechart by merging UC2 first since it is the UC that logically and temporally precedes the other two UCs. Some students wanted to start with UC3, as the main UC from the primary actor perspective, but more students wanted to start with UC2. It appears in retrospect that the students could have started with either UC. In general, one should be able to start with any UC.

Figure 4 shows the turn on event as the first operation of UC2 and as the initial event in the partial unified UC statechart built so far. Figure 4 shows also capturing Operation 2 of UC2 as the accepting events state.

Operation 2 of UC2 was judged by a number of students to be poorly written because:

- it is written in a very generic fashion, i.e., it says that the "System (again) excepts external events", which is not domain-specific functionality, i.e., almost every CBS accepts external events, and

- the term "again" indicates tight coupling with some other UC.

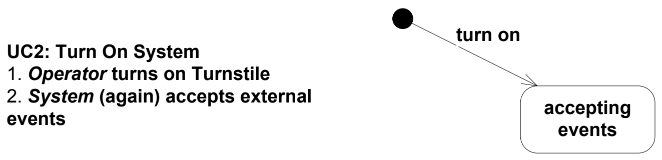

UC2: Turn On System
1. ***Operator* turns on Turnstile**
2. ***System* (again) accepts external events**

Figure 4: Building Unified UC Statechart (1)

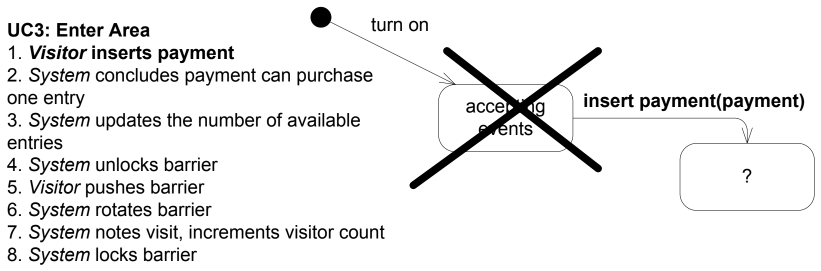

UC3: Enter Area
1. ***Visitor* inserts payment**
2. *System* concludes payment can purchase one entry
3. *System* updates the number of available entries
4. *System* unlocks barrier
5. *Visitor* pushes barrier
6. *System* rotates barrier
7. *System* notes visit, increments visitor count
8. *System* locks barrier

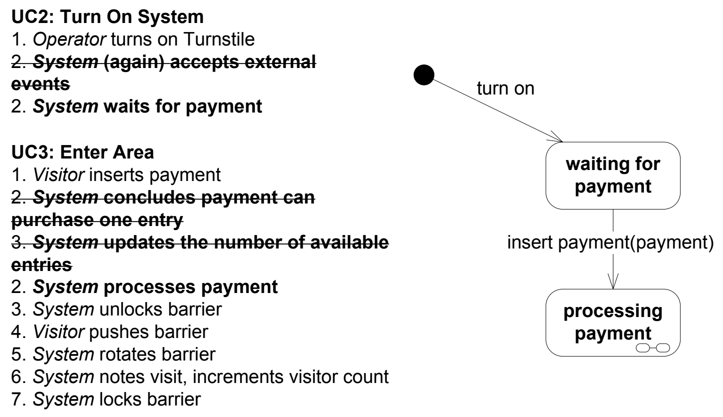
Figure 5: Building Unified UC Statechart (2)

Nevertheless, the students decided not to tackle these problems until they explored and integrated the other UCs.

The next UC that students decided to tackle was UC3. Figure 5 shows how the first external event, insert payment, was integrated into the partial unified UC statechart. The students observed that the state accepting events obtained from Operation 2 of UC2 does not capture the intent of the event of the Operation 2 of UC3 and cannot be merged with the unified UC statechart due to a difference of abstraction levels and concerns, i.e., the first part of Operation 2 of UC2 is a general observation about accepting events, while the second part of Operation 2 of UC3 captures the domain-specific functionality of processing a payment.

The students judged that Operation 2 of UC3 did not capture the CBS's activity, but rather its postcondition. In addition, this postcondition was judged as overly specific due to its specification of exactly one entry. Therefore, the students proceeded by replacing accepting events by waiting for payment in UC2 and modifying Operation 2 of UC3.

Figure 6 shows the modifications of Operation 2 of each UC as well as the modification of Operation 3 of UC3. The students realized that Operation 3 of UC3 was written at the same abstraction level as Operation 2 of UC3 and is thus a part of the same activity. Therefore, Operation 3 of UC3 should be merged with the new Operation 2 of UC3. The students then merged Operations 2 and 3 of UC3 as the activity processing payment, which was captured as a sub-ma-


UC2: Turn On System
1. *Operator* turns on Turnstile
~~2. ***System* (again) accepts external events**~~
2. ***System* waits for payment**

UC3: Enter Area
1. *Visitor* inserts payment
~~2. ***System* concludes payment can purchase one entry**~~
~~3. ***System* updates the number of available entries**~~
2. ***System* processes payment**
3. *System* unlocks barrier
4. *Visitor* pushes barrier
5. *System* rotates barrier
6. *System* notes visit, increments visitor count
7. *System* locks barrier

Figure 6: Building Unified UC Statechart (3)

chine state[13]. The sub-machine state was expected to be decomposed later and was expected to include the activities of the old Operations 2 and 3, among other activities in the decomposition.

Figure 7 shows the refining of the unified UC statechart using a composite state. The students judged the modified Operation 2 of UC2 UC to be at an abstraction level lower than that originally intended for UC2 and to be of a different business concern. Therefore, they decided to introduce a new higher level state controlling access and to modify UC2 and the unified UC statechart appropriately. From that point on, UC2 was considered to be written at a higher abstraction level than UC3. That is, the activities of UC3 became part of the composite activity captured as Operation 2 of UC2.

Figure 8 shows how the students proceeded with the integration of Operation 3 of UC3 into the unified UC statechart. The positioning of the Operation 3, unlocks barrier, immediately after Operation 2, processing payment, was recognized as introducing a big logical gap in the state machine. Missing was the conclusion of a successful payment that results in unlocking the barrier and notifying Visitor. Operation 3 of UC3 was modified to express the conditions under which the barrier is unlocked. The students updated also the system sequence diagram for UC3 to show notification to Visitor, as shown in Figure 9.

Figure 10 shows the refinement of Operation 3 of UC3. Dealing with the successful payment option raised the issue of dealing with an alternative when pay-

[13]That this state is a sub-machine state is indicated by the infinity-like symbol inside the state in Figure 6.

UC2: Turn On System
1. *Operator* turns on Turnstile
~~2. *System* (again) accepts external events~~
~~2. ***System* waits for payment**~~
2. ***System* controls access to restricted area**

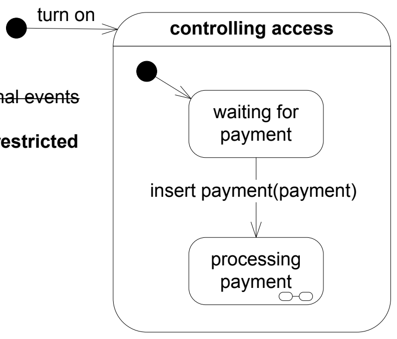


Figure 7: Building Unified UC Statechart (4)

UC3: Enter Area
1. *Visitor* inserts payment
~~2. *System* concludes payment can purchase one entry~~
~~3. *System* updates the number of available entries~~
2. *System* processes payment
~~3. ***System* unlocks barrier**~~
3. ***System* unlocks barrier if payment OK and notifies Visitor**
4. *Visitor* pushes barrier
5. *System* rotates barrier
6. *System* notes visit, increments visitor count
7. *System* locks barrier

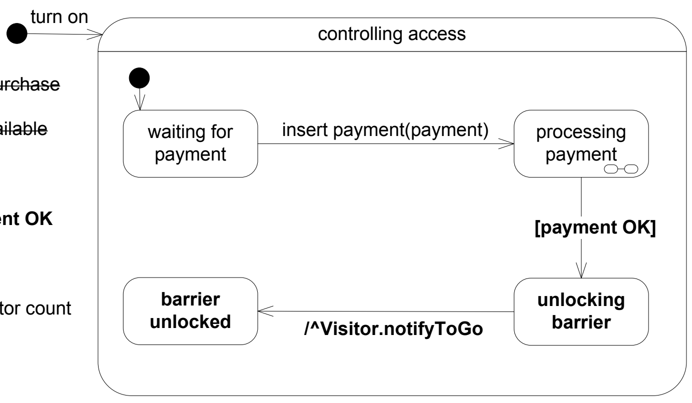


Figure 8: Building Unified UC Statechart (5)

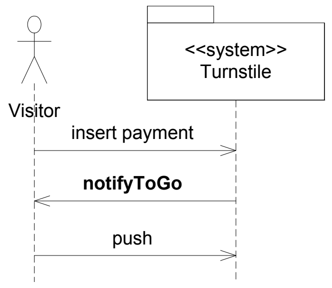
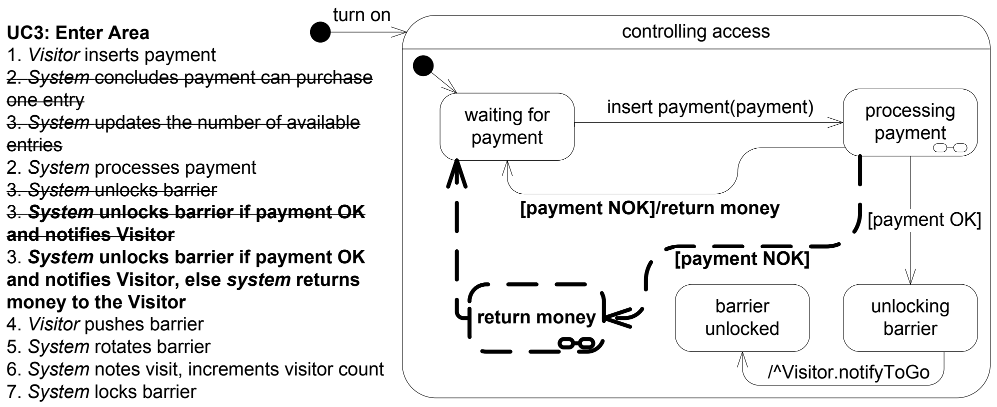


Figure 9: UC3 System Sequence Diagram



UC3: Enter Area

1. *Visitor* inserts payment

~~2. *System* concludes payment can purchase one entry~~

~~3. *System* updates the number of available entries~~

2. *System* processes payment

~~3. *System* unlocks barrier~~

~~3. ***System* unlocks barrier if payment OK and notifies Visitor**~~

3. ***System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor**

4. *Visitor* pushes barrier

5. *System* rotates barrier

6. *System* notes visit, increments visitor count

7. *System* locks barrier

Figure 10: Building Unified UC Statechart (6)

ment is not sufficient and money should be returned. The change was incorporated into a new Operation 3 for UC3. Students initially captured this alternative as an activity shown with the transition and the state drawn with a thick dashed line in Figure 10. Since this unified UC statechart specification was carried out primarily as an educational exercise, for pedagogical expediency, the author decided to consider returning payment to Visitor as a non-decomposable and uninterruptible activity, and thus the author indicated return money as an *action* rather than as an *activity*.

Figures 11 and 12 show the modifications to Operations 4 and 5 of UC3 and their integration into the unified UC statechart. At first, it appeared that Operations 4 and 5 could be integrated in a straightforward fashion as depicted in Figure 11, but analysis of both UC3 and the unified UC statechart exposed a logical problem with having System instead of Visitor rotating Barrier. The students noticed that Barrier is an external entity and should, therefore, be outside of System's boundary. The students modified Operations 4 and 5 as depicted in Figure 12.

The analyses of Operations 4 and 5 and their subsequent integration into the unified UC statechart showed that Barrier should be external to System's boundary. The students therefore decided to redefine System's boundary to exclude Barrier; they updated the context diagram, as shown in Figure 13, and the system sequence diagrams for UC3 and UC1, as shown in Figure 14. As a result of this exclusion, Barrier had become an *actor*. After some discussion, the students realized that they had the wrong actor for UC3, i.e., Visitor instead of Barrier. After some more discussion, the students realized that this change of actor could be ignored at the UC3 level, and the rest of the unified UC statechart was deemed

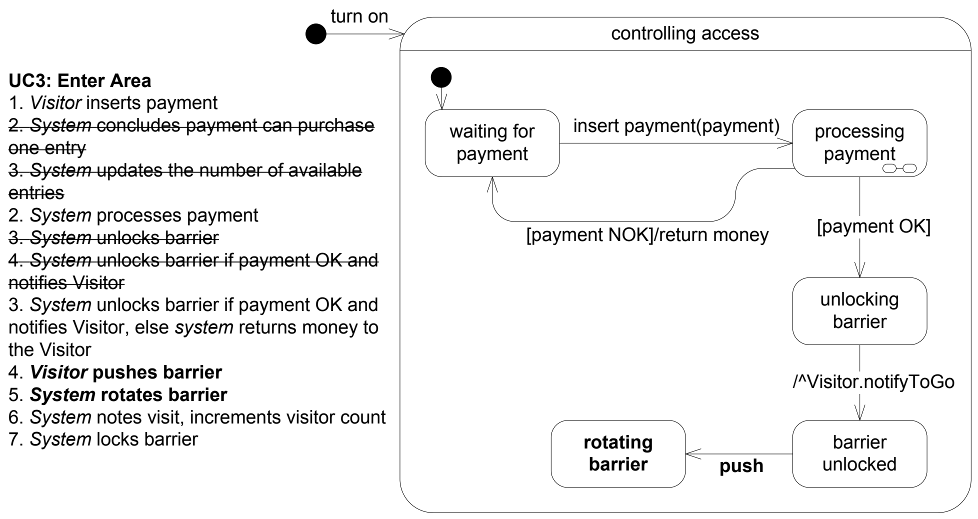

turn on
controlling access
waiting for payment
insert payment(payment)
processing payment
[payment NOK]/return money
[payment OK]
unlocking barrier
/^Visitor.notifyToGo
barrier unlocked
push
rotating barrier
UC3: Enter Area
1. *Visitor* inserts payment
~~2. *System* concludes payment can purchase one entry~~
~~3. *System* updates the number of available entries~~
2. *System* processes payment
~~3. *System* unlocks barrier~~
~~4. *System* unlocks barrier if payment OK and notifies Visitor~~
3. *System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor
4. *Visitor* pushes barrier
5. *System* rotates barrier
6. *System* notes visit, increments visitor count
7. *System* locks barrier

Figure 11: Building Unified UC Statechart (7)

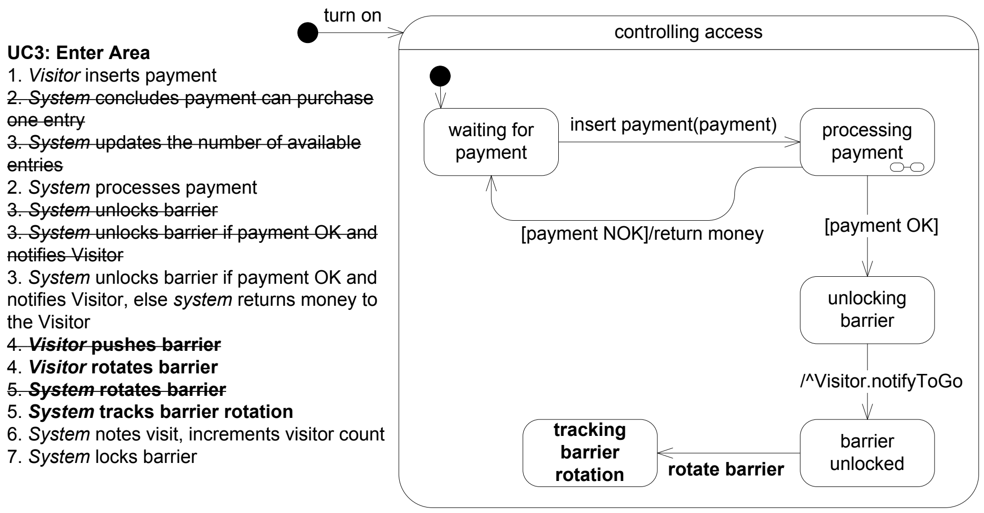

turn on
controlling access
waiting for payment
insert payment(payment)
processing payment
[payment NOK]/return money
[payment OK]
unlocking barrier
/^Visitor.notifyToGo
barrier unlocked
rotate barrier
tracking barrier rotation
UC3: Enter Area
1. *Visitor* inserts payment
~~2. *System* concludes payment can purchase one entry~~
~~3. *System* updates the number of available entries~~
2. *System* processes payment
~~3. *System* unlocks barrier~~
~~3. *System* unlocks barrier if payment OK and notifies Visitor~~
3. *System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor
~~**4. *Visitor* pushes barrier**~~
4. *Visitor* rotates barrier
~~**5. *System* rotates barrier**~~
5. *System* tracks barrier rotation
6. *System* notes visit, increments visitor count
7. *System* locks barrier

Figure 12: Building Unified UC Statechart (8)

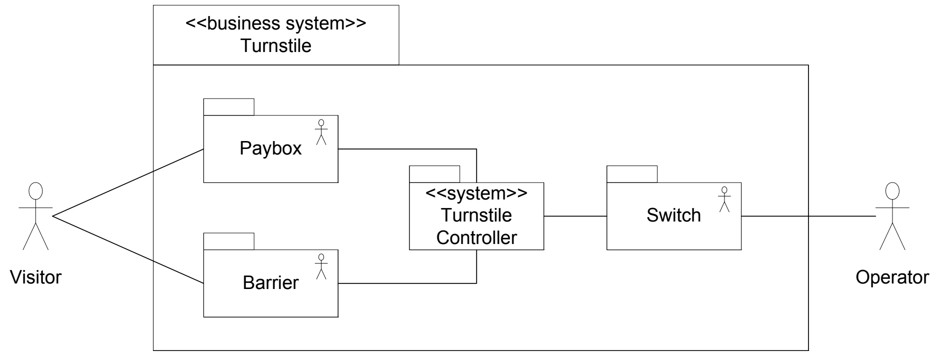


Figure 13: Redefined System Boundary

correct. The reason that the change of actor could be ignored was that the only responsibility of Barrier was to act as a user interface without providing additional complex functionality. The only required change was the additional system outputs, shown in Figure 14, that go to the actors.

So, the students defined a new system boundary in which the CBS under specification is Turnstile Controller. They realized that the old System was in fact a *business system* that included Barrier, Paybox, Switch, and the new System, as is shown in Figure 13. Again, the students decided that there was no need to change the UCs since the *new actors*, Barrier, Paybox, and Switch, acted merely as *user interfaces* between System and the *old actors*, Visitor and Operator, without providing any additional functionality.

Thus, the students moved away from the traditional recommendation of what a UC should capture and how the actors and system boundary ought to be modeled. This movement is not surprising. We have observed the same tendency to ignore actual actors in many other cases, such as using a writer as an actor rather than a keyboard as an actor in a specification of a word processing system.

Figure 15 shows the simplification of the second part of Operation 6 of UC3. The students judged the second part as redundant since it was at a lower abstraction level than the first part of the same operation, the noting visit activity. Some students judged incrementing visitors count to be *a part of* the noting visit activity. Therefore, the noting visit activity needed to be decomposed further during a later refinement of the unified UC statechart.

Figure 16 shows the integration of the last operation of UC3 into the unified UC statechart.

At this point, the students were ready to tackle the last UC, UC1. Figure 17

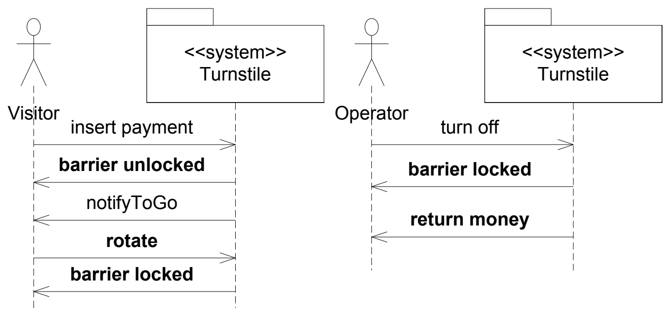


Figure 14: UC3 and UC1 System Sequence Diagrams

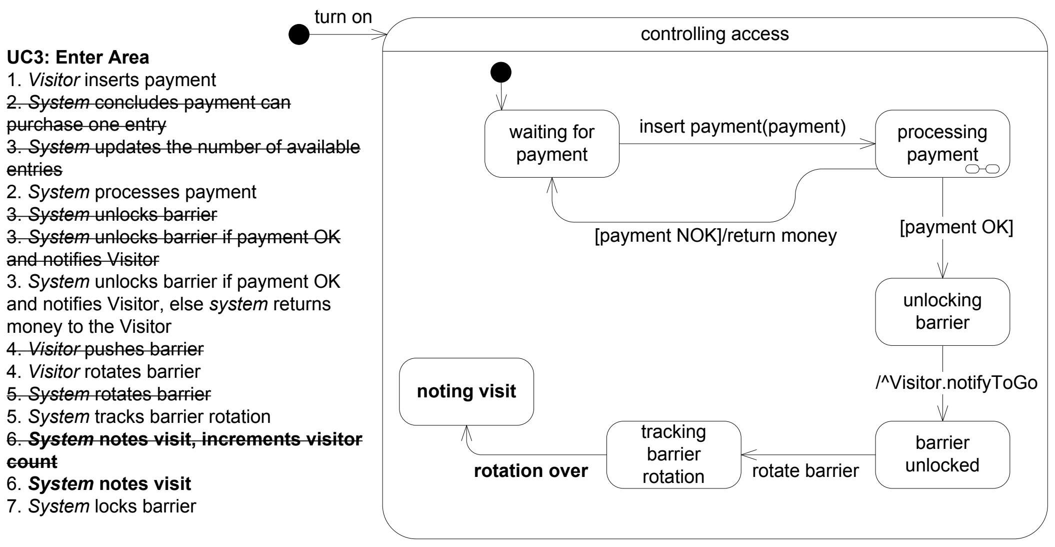

UC3: Enter Area
1. *Visitor* inserts payment
~~2. *System* concludes payment can purchase one entry~~
~~3. *System* updates the number of available entries~~
2. *System* processes payment
~~3. *System* unlocks barrier~~
~~3. *System* unlocks barrier if payment OK and notifies Visitor~~
3. *System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor
~~4. *Visitor* pushes barrier~~
4. *Visitor* rotates barrier
~~5. *System* rotates barrier~~
5. *System* tracks barrier rotation
~~6. ***System* notes visit, increments visitor count**~~
6. ***System* notes visit**
7. *System* locks barrier

Figure 15: Building Unified UC Statechart (9)

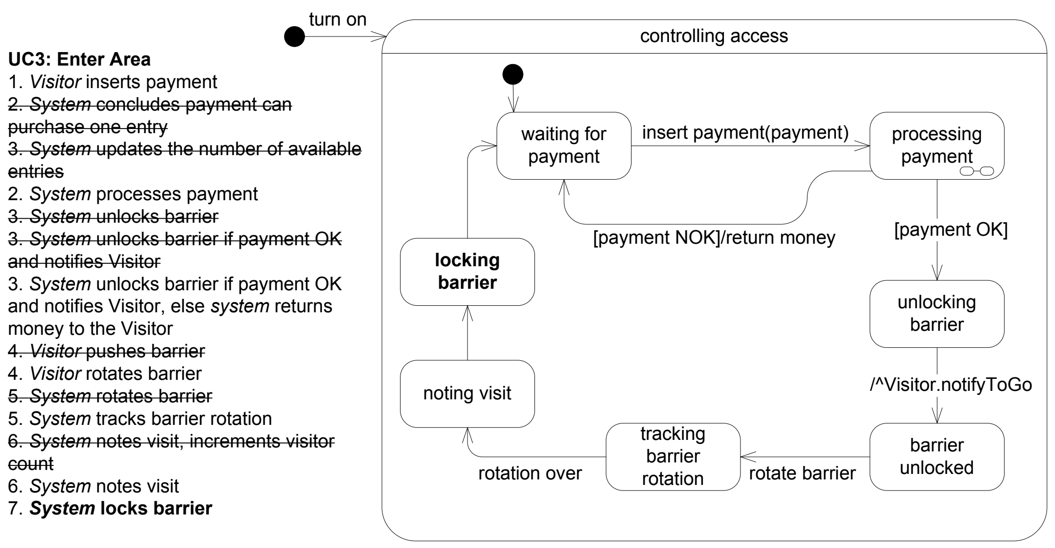

UC3: Enter Area
1. *Visitor* inserts payment
~~2. *System* concludes payment can purchase one entry~~
~~3. *System* updates the number of available entries~~
2. *System* processes payment
~~3. *System* unlocks barrier~~
~~3. *System* unlocks barrier if payment OK and notifies Visitor~~
3. *System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor
~~4. *Visitor* pushes barrier~~
4. *Visitor* rotates barrier
~~5. *System* rotates barrier~~
5. *System* tracks barrier rotation
~~6. *System* notes visit, increments visitor count~~
6. *System* notes visit
7. ***System* locks barrier**

Figure 16: Building Unified UC Statechart (10)

shows integration of Operation 1 of UC1 into the unified UC statechart. Deciding from which state to send the turn off event indicated that the earlier decisions of treating UC2 to be at a higher abstraction level than UC3 and of introducing the composite controlling access state were very useful. The introduction of the composite state would have been required at this stage anyway since the turn off event has to be handled from every state in the unified UC statechart. Therefore, the students captured the turn off event on a transition originating from the envelope of the composite controlling access state.

The students judged Operation 2 of UC1 to contain information at a lower abstraction level than what is captured in the UC2, whose functionality is opposite of that of UC1. Some of the students pointed out also that it was not clear at all from the context what resetting counters for available entries and visitors meant. Therefore, the students decided to move this operation to a higher abstraction level and to postpone its decomposition. They changed Operation 2 to be the resetting activity, which needed further decomposition.

Figure 18 shows that the students judged also Operation 3 to be a part of the resetting activity, and Operation 4 became the new Operation 3 in the modified UC1.

Figure 19 shows that the first part of the new Operation 3 introduced the need for a new state off, while the second part of the new Operation 3 was judged as

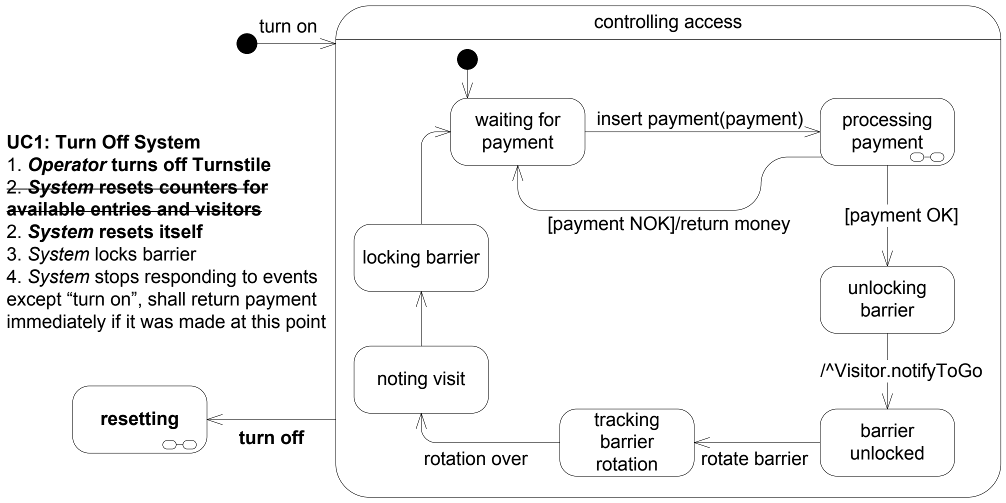
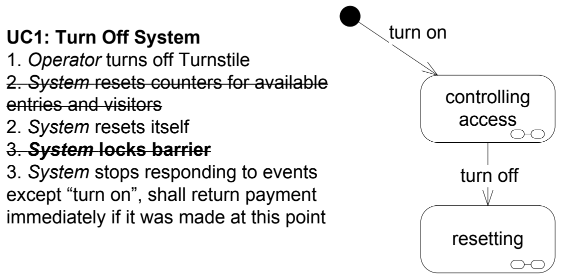


Figure 17: Building Unified UC Statechart (11)



UC1: Turn Off System
1. Operator turns off Turnstile
~~2. System resets counters for available entries and visitors~~
2. System resets itself
~~3. System locks barrier~~
3. System stops responding to events except "turn on", shall return payment immediately if it was made at this point
turn on
controlling access
turn off
resetting

Figure 18: Building Unified UC Statechart (12)

UC1: Turn Off System
1. *Operator* turns off Turnstile
~~2. *System* resets counters for available entries and visitors~~
2. *System* resets itself
~~3. *System* locks barrier~~
~~3. ***System* stops responding to events except "turn on", shall return payment immediately if it was made at this point**~~
3. ***System* turns off**

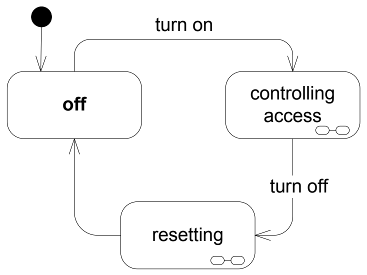


Figure 19: Building Unified UC Statechart (13)

UC2: Turn On System
1. *Operator* turns on Turnstile
~~2. *System* (again) accepts external events~~
~~2. *System* waits for payment~~
~~2. *System* controls access to restricted area~~
2. ***System* sets up**
3. ***System* controls access to restricted area**

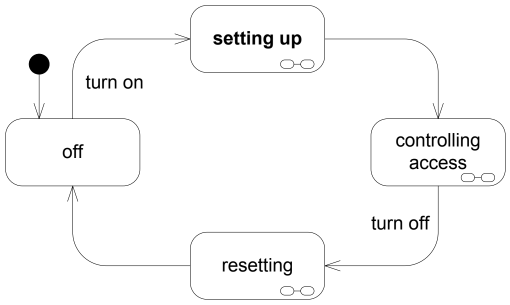


Figure 20: Building Unified UC Statechart (14)

redundant and possibly a part of the resetting activity that was to be decomposed later.

Finally, the resetting activity in the UC1 exposed the need for a corresponding setting up activity in UC2, as shown in Figure 20.

Figure 21 shows the final, integrated unified UC statechart of all three UCs.

6 Evaluation of UCUM

The evaluation of the effectiveness of UCUM was carried out through three case studies, the TCS and two additional case studies. The second case study of UCUM use, the Elevator case study (ECS), concerns 12 SRSs for a medium-sized controller CBS for two-elevators in a low-rise building. Each SRS was produced by one CS846 graduate student working independently. Rather than working from a fictitious project description, students were required to analyze an already deployed elevator system, to ensure that all the students had a common starting point. Moreover, any ambiguity, which might exist in a fictitious project descrip-

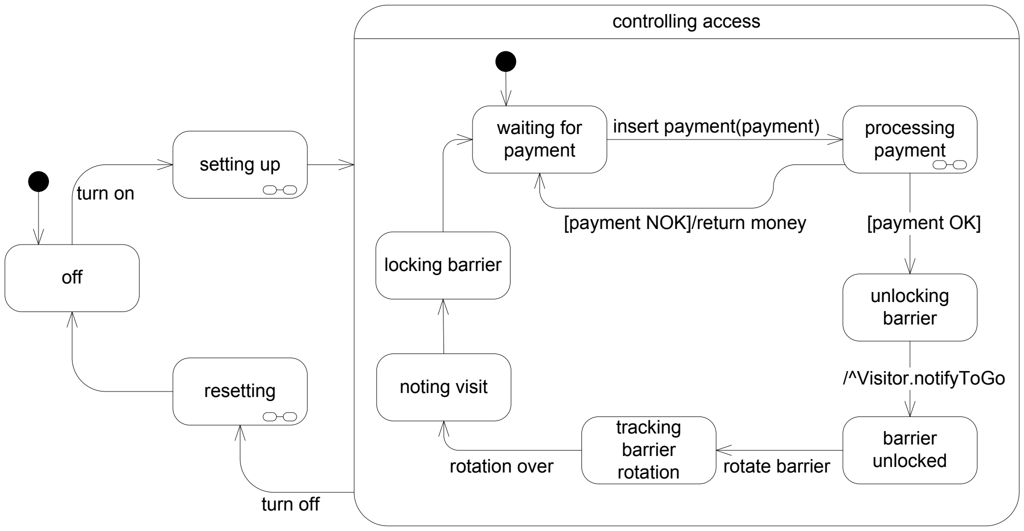


Figure 21: Final Unified UC Statechart

tion, could be resolved by observing the actual elevators' behavior. The effort required for a good specification of this system is approximately 100 person hours. This time estimate is based on the discussions with students and comparison of the results.

Each student handed in two partial SRSs before handing in the final SRS. Each SRS was required to show a specific set of artifacts. The first partial SRS had to show the initial set of UCs for the CBS. Each student was allowed to see all the students' sets of UCs before handing in the second partial SRS so that each student could have as good a set of UCs as possible before constructing the unified UC statechart. However, thereafter, no student was allowed to see any other's work. The complete set of partial and final SRSs can be found at the CS846 course website[14].

The third case study of UCUM use, the VoIP case study (VCS), concerns 46 SRSs for a large-sized VoIP CBS. The description of the CBS can be found at the course website[15]. The VCS was carried out over two terms of CS445. Each SRS was produced by a group of 3 or 4 primarily undergraduate CS445 students working together, with 4-member groups being in the majority. We estimated the effort required for a good specification of this system to be approximately 400

[14] `http://se.uwaterloo.ca/~dberry/ATRE/ElevatorSRSs/`
[15] `http://se.uwaterloo.ca/~dberry/cs445w06`

person hours.

In the first term, each group handed in two partial SRSs before submitting its final SRS; also, two weeks before the final SRS was due, each group led a formal walkthrough of its work in front of another group and a TA. In the second term, the groups were not required to hand in the first partial SRS, but each group had to perform a formal walkthrough to the TA and course staff demonstrating the UCs they had found so far. In both terms, each successive partial and final SRS was required to show a growing set of specific artifacts. Each group worked independently, and no group was allowed to see any other group's work except during the formal walkthroughs. Each group worked with its own TA, who served as its customer in a simulated customer–analysts relationship.

The results of the ECS and the VCS are mostly positive, but several problems were noted. The negative results, which concern mostly working with large CBSs, are explained with examples from the VCS. Thus, the VCS, with the largest CBS among the three case studies, served as a real test for the usefulness of the unified UC statechart and for the effectiveness of UCUM.

6.1 Positive Results

We observed nine positive results altogether coming from the *process* of unifying the UCs into a unified UC statechart. Each of the first seven positive results is only that *when* the unification was being done, it was *easier than in the past for an analyst to* do something beneficial. It would be improper to state the results more strongly. Therefore, the statement of each of these positive results should begin with "When unifying UCs of a CBS into a unified UC statechart, it was easier than in the past for an analyst to". Since this long phrase would be repeated seven times, to save some space, we abbreviate it as: "Unification Helps To".

The first five of these positive results were observable in the SRSs of all three case studies, while the last four were observable in the SRSs of only the ECS and the VCS, those with the larger CBSs. Therefore, the first five positive results and, thus, the general usefulness of building a unified UC statechart from a CBS's UCs do not depend on the size of the CBS.

The nine positive results are:

1. Unification Helps To *identify the boundary of the CBS under specification.* The set of actors and UCs for a CBS both depend on and help determine the CBS's boundary. Therefore, the full set of actors and UCs for a CBS cannot be known until the CBS's boundary is known. Conversely, until all

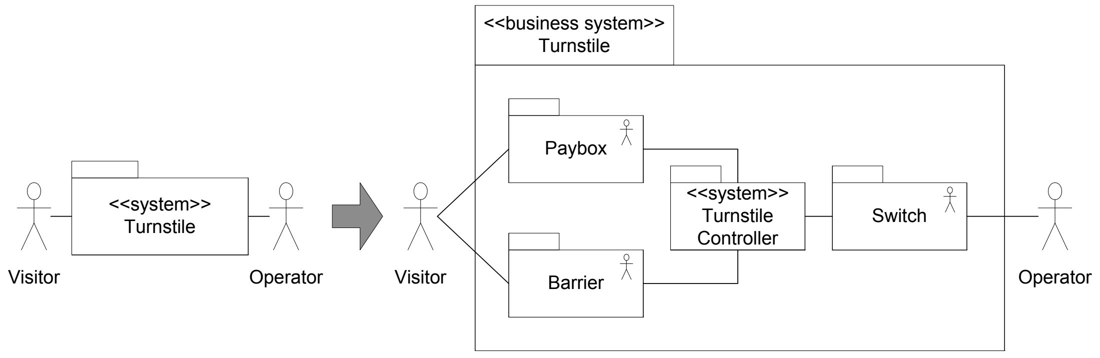


Figure 22: CBS Boundary Change

of a CBS's actors and UCs are known, it is hard to define the CBS's exact boundary. Defining the boundary is even harder when there are multiple analysts and multiple stakeholders each with a different perception of the CBS's boundary. Even for the small Turnstile, the boundary established in the SRS[16] from which we took the initial 3 UCs proved to be wrong. The correct boundary, as discovered through UCUM, is as shown in Figure 22. Interestingly, for the elevator controller CBS, many a student found that the CBS's boundary should be around the controller hardware and software, excluding other devices with which the passenger interacts, e.g., elevator cab, buttons, etc. In other words, the actors implied by the tighter boundary were the devices that serve as interfaces between passengers and the controller. It is irrelevant to the controller what or who causes a button to be pushed. Even after learning about the tighter boundary, many a student made a conscious decision to stick with the traditional boundary around the passengers, even though a passengers never touches the controller.

2. Unification Helps To *identify abstraction level clashes and redundant operations in the UCs.* Correcting the abstraction levels of UCs is necessary to successfully unify the UCs of a CBS into a unified UC statechart. For example, in the original SRS for the Turnstile, the abstraction level of the UC Turn Off System was inconsistent with that of its opposite, the UC Turn On System. Turn Off System's definition gave low-level details, such as resetting counters and Turn On System's definition was written at a higher abstraction level with no reference to internals. The solution was

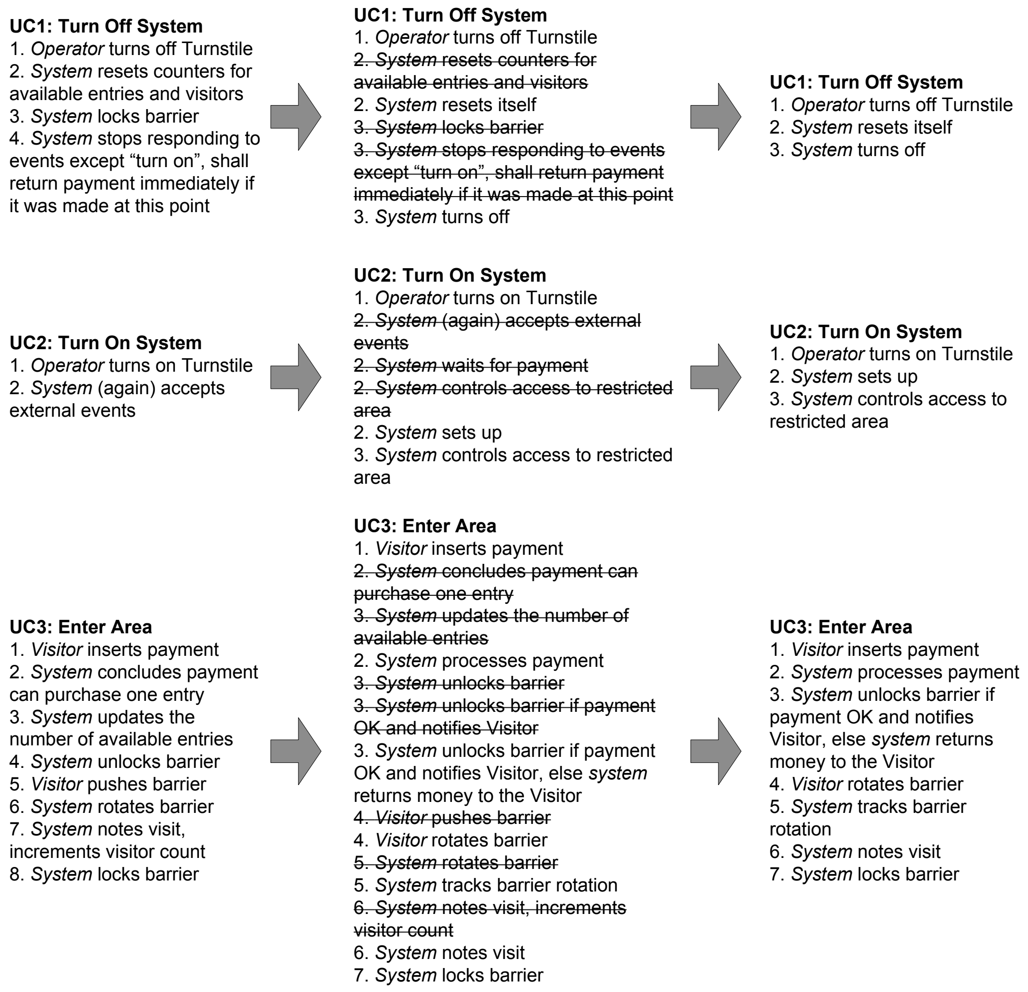
[16] `http://se.uwaterloo.ca/~dberry/cs445w06/turnstilesystem.pdf`

to write each UC with no reference to internals. Each decomposition was left to appear in the unified UC statechart.

3. Unification Helps To *identify incorrect ordering of operations in a UC's description.* It is often the case that a UC's operations are out of order, because of the scope of the UC or the informality of UC description. For example, in the original SRS for the Turnstile, in Operation 4 of the UC Turn Off System, the payment was being returned *after* the CBS was shut down. Moreover, it was not even certain that the payment *should* be returned. The problem was diagnosed as the analyst's having detected an exception and having inserted the exception too quickly into a random operation. Including this exception into the unified UC statechart proved to be awkward, if not impossible, and more analysis was needed to find its rightful place.

4. Unification Helps To *detect missing functionality among the UCs.* While the first three results address *consistency* of UCs, the fourth result addresses *completeness* of UCs from each actor's perspective. Detecting missing functionality requires domain expertise, and even then it is hard. Any requirements analysis method can help but not guarantee that the SRS will describe all needed functionality. Since one can never be certain when the last function is found, it is hard to know how complete an SRS is. Nevertheless, it appears that in each of the case studies, unifying the UCs of the CBS into a unified UC statechart did help expose functions of the CBS that were missing in the UCs. The kind of rework that appears in the middle column of Figure 23 is typical. The left column of Figure 23 shows the original UCs of the Turnstile *before* specification of the unified UC statechart, the middle column shows modifications to the UCs *during* specification of the unified UC statechart, and the right column shows UCs *after* the unified UC statechart was completed. An observable weakness of the standard UC-driven requirements analysis methods is the lack of a way detect functionality needed to support concurrency among UCs. One way to detect this kind of functionality is to attempt to integrate the UCs, exactly what unification of the UCs into a unified UC statechart is doing, and is doing before coding begins.

5. Unification Helps To *simplify the descriptions of UCs.* This result is a natural consequence of the first four. Building a unified UC statechart almost universally led to simplifying and clarifying the descriptions of the UCs that were being unified. In many a case, an operation whose description was a



UC1: Turn Off System
1. *Operator* turns off Turnstile
2. *System* resets counters for available entries and visitors
3. *System* locks barrier
4. *System* stops responding to events except “turn on”, shall return payment immediately if it was made at this point

UC1: Turn Off System
1. *Operator* turns off Turnstile
~~2. *System* resets counters for available entries and visitors~~
2. *System* resets itself
~~3. *System* locks barrier~~
~~3. *System* stops responding to events except “turn on”, shall return payment immediately if it was made at this point~~
3. *System* turns off

UC1: Turn Off System
1. *Operator* turns off Turnstile
2. *System* resets itself
3. *System* turns off

UC2: Turn On System
1. *Operator* turns on Turnstile
2. *System* (again) accepts external events

UC2: Turn On System
1. *Operator* turns on Turnstile
~~2. *System* (again) accepts external events~~
~~2. *System* waits for payment~~
~~2. *System* controls access to restricted area~~
2. *System* sets up
3. *System* controls access to restricted area

UC2: Turn On System
1. *Operator* turns on Turnstile
2. *System* sets up
3. *System* controls access to restricted area

UC3: Enter Area
1. *Visitor* inserts payment
2. *System* concludes payment can purchase one entry
3. *System* updates the number of available entries
4. *System* unlocks barrier
5. *Visitor* pushes barrier
6. *System* rotates barrier
7. *System* notes visit, increments visitor count
8. *System* locks barrier

UC3: Enter Area
1. *Visitor* inserts payment
~~2. *System* concludes payment can purchase one entry~~
~~3. *System* updates the number of available entries~~
2. *System* processes payment
~~3. *System* unlocks barrier~~
~~3. *System* unlocks barrier if payment OK and notifies Visitor~~
3. *System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor
~~4. *Visitor* pushes barrier~~
4. *Visitor* rotates barrier
~~5. *System* rotates barrier~~
5. *System* tracks barrier rotation
~~6. *System* notes visit, increments visitor count~~
6. *System* notes visit
7. *System* locks barrier

UC3: Enter Area
1. *Visitor* inserts payment
2. *System* processes payment
3. *System* unlocks barrier if payment OK and notifies Visitor, else *system* returns money to the Visitor
4. *Visitor* rotates barrier
5. *System* tracks barrier rotation
6. *System* notes visit
7. *System* locks barrier

Figure 23: Turnstile UC Changes

full paragraph of text was replaced by a single sentence description. Clearly defining goals and activities during construction of a unified UC statechart exposed overly complex descriptions of UCs. Simplifying UC descriptions in turn allowed easier identification of goals, activities, inputs, outputs, and other data.

6. Unification Helps To *see how to restructure the descriptions of the UCs.* We saw that many a student restructured the descriptions of her UCs after finishing the unified UC statechart. Typically, the student used pseudocode to describe UC operations, particularly for iterative and alternative paths. We interpreted this restructuring to be a positive result because the restructuring helped the student to detect often-overlooked alternative paths. The use of pseudocode might be considered a negative, but we feel that the positive of finding more alternative paths outweighs this negative.

7. Unification Helps To *detect opportunities for concurrent UC execution.* Recall that the fourth positive result is that unification allows detection of missing functionality among the UCs and that among the missing functions detected was the support needed for concurrent execution of UCs. Still it is necessary to be able to detect *opportunities* for concurrent execution of UCs. The act of unifying UCs into a unified UC statechart shows clearly which UCs can be unified temporally, i.e., can be executed concurrently.

8. *The benefits of unifying UCs of a CBS into a unified UC statechart are independent of the exact method by which the unification is done.* Each student seemed to gain the benefit of the unification no matter which variation of UCUM she used. Some wrote UC statecharts for the UCs before unifying the UCs into a unified UC statechart, and some unified directly from the original UC descriptions. Regardless of how a student or group did the unification, the resulting unified UC statecharts and the resulting SRSs were significantly better than those produced in the CS445 course prior to the introduction of UCUM.

9. The average grade for a CS445 group's VoIP SRS in any UCUM-using term was the same as it was in any previous, non-UCUM-using term, despite that the evaluation criteria were higher and the marking was stricter in the UCUM-using terms. Thus, we found that *the UCUM-assisted SRSs were overall of higher quality than non-UCUM-assisted SRSs*.

Manifestations of the first five positive results can be seen in the refinement, shown in Figure 23, of the three initial Turnstile UCs. As much scrutiny as these UCs had from us and about 140 students, there are still some unresolved problems. For example, it is not clear which actors' goals are served by either the Turn On System or the Turn Off System UC. Also, Operation 3 of the Turn Off System UC is a postcondition rather than the activity it should be. These two examples make it clear that

1. one can never be certain about the quality of UCs, and
2. while unifying UCs into a unified UC statechart does help find problems in the UCs, it cannot guarantee finding all of them.

Even with a very small CBS such as the Turnstile and even with about 140 persons analyzing it, it was not possible to fix all problems caused by the initial choice of UCs. Of course, abandoning the initial choice of UCs might lead to better fixes, but in our experience, just *finding* a problem can be *harder* than fixing it. Fortunately, a benefit of unifying UCs into a unified UC statechart is that it helps the analyst to *find* problems.

In summary, all the positive results and the deepened understanding of the domain can be attributed to the the ability of a unified UC statechart to provide a big picture of a domain model more systematically and more formally than is possible with only UCs. Nevertheless, not all results were positive.

6.2 Negative Results

This subsection discusses pernicious and persistent problems that remain despite all our best efforts. First, this subsection discusses the problems that we believe can be resolved. Then, it discusses problems that we believe are inherent to the method itself and as such cannot be fixed.

The first problem that we observed is *the difficulty of determining what about a CBS should be modeled.* We explained that subsystems, devices, user interface screens, and so on, should not be modeled in the states of a unified UC statechart. Nevertheless, an occasional student did include in his unified UC statechart what was not covered by agreed upon unified UC statechart semantics. Despite the grade penalty for inclusion of non-conventional unified UC statechart states, many of those penalized continued to do it. Therefore, it would be profitable to determine *why* anyone was getting bogged down in details that are irrelevant at the UC level.

The second problem that we observed is *the lack of direct support in unified UC statecharts for representation of concepts and objects*. UCUM is supposed to be a part of an object-oriented domain analysis method for our course projects. As such, conceptual analysis is supposed to follow the completion of the unified UC statechart. One of the analysis steps is assigning to concepts the activities captured during unification to the unified UC statechart. How to represent this assignment of activities to concepts was left to the students to figure out. Some used comments, some extended activity names to include responsible concepts, etc. In any case, none of these representations is a part of the standard SC notation. We did observe the tendency of a typical student to include in her unified UC statechart some high-level conceptual analysis constructs such as subsystems. This tendency suggests the usefulness of extending SC notation with some conceptual analysis notation, as suggested by Glinz [7].

The third problem that we observed is *the difficulty of unifying UCs one by one in some cases*. Many a student claimed that it was easier

1. to grasp all UCs together and then to build the unified UC statechart than
2. to unify UCs one by one into a growing unified UC statechart in either of the two ways suggested by UCUM.

We suspect that this preference comes from the typical low quality of UC descriptions. Many UC descriptions were poorly structured, with ill-defined CBS boundaries, and with actors missing. Consequently, it was very difficult to build SCs for the UCs. It appears that many a student was simply discouraged by the perceived effort to redo all the UC descriptions, and jumped directly to producing a unified UC statechart, which turned out to require lots of rework. We say "perceived effort", because in the end, the thinking needed to fix the poor unified UC statechart was the same as would be needed to redo the UC descriptions. The typical directly produced unified UC statechart had a large number of inconsistencies with use cases, was more difficult to refine, and was at much higher abstraction level than the typical unified UC statechart produced by unifying UCs one by one. Occasionally, the directly produced unified UC statechart was at such a high abstraction level that its nodes represented use case names, and these nodes were not decomposed any further. As a consequence, we strongly believe that UCs should be unified one by one into a growing unified UC statechart, as it was suggested to the students in the first place.

The fourth problem that we observed is *that some students could not fix all of their UCs due to time limitations*. This problem is related to the third, namely that building a unified UC statechart can require completely redoing all UC descriptions. Many a student, who had written very poor UC descriptions and postponed unified UC statechart specification, simply did not have the time to redo all his UC descriptions. To solve this problem, we need to give students a clear indication on how much effort it takes to build a unified UC statechart and what the impact of poor UC specifications can be.

Finally, we describe the problems that we believe are inherent to UCUM and therefore not fixable.

The main negative effect of unifying UCs of a CBS into a unified UC statechart for the CBS is *the additional effort that has to be invested as a result of the steep learning curve and the inherent difficulty of specifying behavior*. We saw the additional effort only with the VoIP CBS because we could compare the effort spent by students writing SRSs of the VoIP CBS in UCUM-using terms with the effort spent by writing SRSs of the same VoIP CBS in previous, non-UCUM-using terms.

Teaching students and TAs UCUM required 4 hours that were not originally allocated to the course. Of these 4 hours, 2 were spent teaching unified UC statechart unification and 2 were spent teaching how unification fits in the overall RE process. An additional hour was set aside for a question-and-answer session about the material. The head TA, Svetinovic, responsible for answering students' questions found his workload increased about 30% over that in previous terms, in which UCUM was not used.

Each term in CS445, we have each TA report his or her actual workload for the course. As a result, we are able to say that the average number of meetings in a term between a group and its TA, as analysts and customer, increased from about 6–8 in previous non-UCUM-using terms to about 10 in the UCUM-using term. That is, learning and using any variant of UCUM required about 25% more elicitation effort. Because we had anticipated at the beginning of the first UCUM-using term that UCUM might require more work, we switched from encouraging 3-person groups to encouraging 4-person groups. In retrospect, the increased specification workload for UCUM is proportional to the increase in group size.

The increased workload was not without benefit, namely in the observed increased overall quality of the SRSs that the groups produced. In particular, the typical group elicited more requirements along the way than in the past.

The other negative result is *the continued difficulty of dealing with multiple processes and object concurrency*, a difficulty not really addressed by any existing method. That this difficulty remains with UCUM is disappointing because unified UC statecharts are supposed to explicitly expose opportunities for concurrency

[4], and indeed the seventh positive result was that Unification Helps To detect opportunities for concurrent UC execution. However, the only concurrency that is detected is among the UCs. More general concurrency, e.g., among processes and objects, remains hidden. A possible approach for more complete concurrency detection is merging the SC notation with others to build more general models that expose concurrency opportunities better, as suggested by Glinz [7].

7 Related Work

This section compares the work of this paper to that of the three papers that inspired UCUM, namely papers by Glinz, Whittle *et al.*, and Harel *et al.* [6, 26, 11]. Each of these papers describes one formal treatment of unification of UCs into a statechart similar in semantics to our unified UC statechart[17]. Several others, including Somé *et al.* [21], van Lamsweerde *et al.* [25], Khriss *et al.* [13], Somé [22], and Damas *et al.* [3] describe algorithms and methods for synthesizing various domain models, including one in the statecharts notation, from UCs. Detailed comparisons of the three methods on which UCUM is based, among a number of other scenario and statechart unification methods, are offered by Saiedian *et al.* [20] and by Liang *et al.* [16].

Glinz presents a method, intended to be automated, of constructing a statechart expression of the domain model of a CBS from a set of statecharts, one for each UC of the CBS. During the construction, whenever an inconsistency shows up, e.g., two transitions from one state going to two different states under the same event, the original UC statecharts must be modified. Glinz's plan was to automate the construction so that analysis, including checking for inconsistencies, can be automated as well.

Harel *et al.* describe an algorithmic method to synthesize a statechart expression of a domain model of a CBS from a set of live sequence charts (LSCs), one for each UC of the CBS. LSCs are formally defined enhancements of sequence diagrams (SDs) with precise semantics, the ability to define existential or universal UCs, and specified preconditions. Their algorithm has been implemented as part of a tool that animates LSCs. When the algorithm fails, due to inconsistencies among input LSCs, the user is expected to correct the problems in the LSCs.

Whittle *et al.* describe an algorithmic method to generate a statechart expression of a domain model of a CBS from a set of SDs, one for each UC of the CBS.

[17]Each of the works described in this chapter uses the term "scenario" for what we call "UC".

Whittle *et al.* have implemented the algorithmic method in a tool. The tool requires user assistance, particularly when the tool detects an inconsistency among the input SDs. The user's response is to change one or more SDs; to change parts of the statechart expression of the domain model that are outside the SDs, e.g., data and preconditions; or both.

Comparisons between the steps, restrictions, and problems in the methods and algorithms of Glinz, Whittle *et al.*, and Harel *et al.* and those of UCUM are similar to comparisons between other pairs of automated and manual processes. Moreover, the benefits that they observe of their methods and algorithms are consistent with the benefits that were observed of UCUM. Thus, it can be said that this work and their work constitute independent confirmations of each other.

8 Conclusion

This paper describes the results of three case studies that evaluate a practical method for unifying UCs of a CBS into a unified UC statechart for the CBS that can be used as part of a RE process to produce a SRS for the CBS. The method was iteratively prototyped through in-course uses of variants of the method. Thus, the case studies both refined the method and validated the usefulness of the unified UC statechart and the effectiveness of the method both to improve the starting UCs and to yield a quality unified UC statechart that becomes part of a quality SRS.

The TCS, the ECS, and the VCS have demonstrated the usefulness and practicality of UCUM, a method similar to the UC unification methods described by Glinz, Whittle *et al.* and Harel *et al.*. Moreover, UCUM has been used on CBSs of relatively large size and has been carried out by a large number of students lacking expertise in statecharts and domain modeling. The three case studies have shown UCUM to provide specific practical benefits to the analysts who apply it, and have exposed the drawbacks of the method.

Specifically, we found that unifying UCs of a CBS into a unified UC statechart, and use of UCUM in particular, makes it easier than in the past for an analyst to:

1. identify the boundary of the CBS,
2. identify abstraction level clashes and redundant steps in the UCs,
3. identify incorrect ordering of operations in a UC's description,
4. detect missing functionality among the UCs,

5. simplify the descriptions of UCs,
6. see how to restructure the descriptions of the UCs, and
7. detect opportunities for concurrent UC execution.

We found also that the benefits of unifying UCs of a CBS into a unified UC statechart are independent of the exact method by which the unification is done, and that UCUM-assisted SRSs were overall of higher quality than non-UCUM-assisted SRSs. In addition, we found a number of drawbacks of using UCUM.

A case study of an actual method use can measure the cost of applying the method. In particular, the three case studies have shown that adding to RE the UCUM way of unifying UCs of a CBS into a unified UC statechart for the CBS increases the cost of requirements elicitation and the subsequent analysis by about 25%. Recall that our student subjects were judged to be equivalent to recently graduated software engineers. Therefore, because of the relative modeling inexperience of the student analysts in the three case studies, this cost increase is probably near a worst-case upper bound.

It is true that performing a unification completely manually forces continual reexamination of the UCs. However, having a tool with picky restrictions on the expression of the input UCs forces more precision in the descriptions of UCs. Perhaps, it is the case that the students of the case studies, having heavily sweated manual unification would greatly appreciate both either of the Whittle *et al.* or the Harel *et al.* tool and the discipline required to prepare the input to the tool.

Finally, subject to the limitations described in Section 3.1, the three case studies support acceptance of the hypothesis raised at the very end of Section 2.

In the future, we would like to perform additional case studies and controlled experiments. In particular, we want to focus on measuring the cost of performing UCUM and the scalability of UCUM.

Acknowledgments

We thank all students and TAs who participated in the case studies and who provided feedback. We thank all reviewers of all earlier versions of this paper for their helpful comments. Davor Svetinovic's work was supported in parts by Canadian NSERC Postgraduate Scholarship PGS B-255929-2002 and a Canadian FCAR Doctoral Scholarship. Daniel Berry's, Michael Godfrey's, and Nancy

Day's work was supported in part by Canadian NSERC Grant Numbers NSERC-RGPIN227055-00, NSERC-RGPIN217236-99, and NSERC-RGPIN240537-01, respectively.

References

[1] Rolv Bræk and Øystein Haugen. *Engineering real time systems: an object-oriented methodology using SDL.* Prentice Hall International, 1993. ISBN 0-13-034448-6.

[2] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Reading, MA, 2000.

[3] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 197–207, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-468-5. doi: http://doi.acm.org/10.1145/1181775.1181800.

[4] Bruce Powel Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-49837-5.

[5] Martin Glinz. Statecharts for requirements specification - as simple as possible, as rich as needed. In *Proceedings of the ICSE2002 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2002.

[6] Martin Glinz. An integrated formal model of scenarios based on statecharts. In *Proceedings of the 5th European Software Engineering Conference*, pages 254–271, London, UK, 1995. Springer-Verlag. ISBN 3-540-60406-5.

[7] Martin Glinz, Stefan Berner, and Stefan Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002. ISSN 0306-4379.

[8] Hassan Gomaa. Designing concurrent, distributed, and real-time applications with UML. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 737–738, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[10] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/42411.42414.

[11] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Lecture Notes in Computer Science*, volume 3393 of *LCNS*, pages 309–324. Springer-Verlag, January 2005.

[12] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[13] Ismail Khriss, Mohammed Elkoutbi, and Rudolf K. Keller. Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams. In *UML'98: Selected papers from the First International Workshop on The Unified Modeling Language UML'98*, pages 132–147, London, UK, 1999. Springer-Verlag. ISBN 3-540-66252-9.

[14] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.

[15] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, NJ, second edition, 2001.

[16] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-394-8. doi: http://doi.acm.org/10.1145/1138953.1138956.

[17] Susan Lilly. Use case pitfalls: Top 10 problems from real projects using use cases. In *Proceedings Technology of Object-Oriented Languages and Systems*, pages 1974–183, Washington, DC, USA, 1999. IEEE Computer Society.

[18] John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1): 31–37, 1999.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, second edition, 2004.

[20] Hossein Saiedian, Prabha Kumarakulasingam, and Muhammad Anan. Scenario-based requirements analysis techniques for real-time software systems: a comparative evaluation. *Requir. Eng.*, 10(1):22–33, 2005. ISSN 0947-3602. doi: http://dx.doi.org/10.1007/s00766-004-0192-6.

[21] Stéphane Somé, Rachida Dssouli, and Jean Vaucher. From scenarios to timed automata: Building specifications from users requirements. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, pages 48–57, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7171-8.

[22] Stéphane S. Somé. Supporting use case based requirements engineering. *Information & Software Technology*, 48(1):43–58, 2006.

[23] Davor Svetinovic, Daniel M. Berry, and Michael Godfrey. Concept identification in object-oriented domain analysis: Why some students just don't get it. In *Proceedings of the IEEE International Conference on Requirements Engineering RE'05*, pages 189–198, 2005.

[24] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.

[25] Axel van Lamsweerde and Laurent Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. Softw. Eng.*, 24(12):1089–1114, 1998. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.738341.

[26] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 314–323, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-206-9. doi: http://doi.acm.org/10.1145/337180.337217.