Domain System StateCharts: The Good, the Bad, and the Ugly

Davor Svetinovic, Daniel M. Berry, Nancy A. Day, Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{*dsvetino,dberry,nday,migod*}*@uwaterloo.ca*

Abstract

This paper presents the results of case studies evaluating our method of unifying use cases (UCs) to derive a unified StateChart (SC) model of the behavior of the domain system (DS) of a proposed computer-based system. An evaluation of the unification method, the obtained SC model of the DS, 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 46 software requirements specifications produced by 149 upper-year undergraduate and graduate students. The results of our 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, who have developed formal treatments of unifying UCs using SCs and, in two cases, have built tools implementing their treatments.

1. Introduction

Analyzing and specifying the behavior of a computer based system (CBS) to be built is a very hard requirements engineering (RE) task. Teaching people to do this task is even harder. In use-case-driven requirements analysis methods [12, e.g.,], the first task an analyst performs in modeling the behavior of the CBS being built is to write use cases that describe the CBS's intended behavior. From these use cases (UCs), he or she begins to model the entire CBS with the goal of eventually writing a software requirements specification (SRS) describing the CBS.

During the last six years, we have observed the production of and have evaluated SRSs, including UC specifications, produced by over 1000 students analyzing and specifying a large voice-over-IP (VoIP) telephony system and its related account-management system [16]. Each of these CBSs was specified using an iterative, UC-driven method. The problems we saw students having motivated the work described in this paper. Also, any statement in this paper about methods comes from these observations.

UCs describe uses of a CBS from the users' perspectives. Domain experts and analysts together typically capture UCs during and after requirements elicitation from many stakeholders, each with a different perspective. We have seen that the typical result of this 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 due to the difficulty of abstracting from multiple UCs to the *big picture* of the domain system. These observations are consistent with those of other authors [13, e.g.,]. In short, the set of UCs is not *good*. Thus, specifying ***good*** UCs is *hard*.

Specifying ***good*** UCs is also *necessary* because of their central role in UC-driven requirements analysis methods. In these methods, UCs drive all subsequent analysis, design, and coding. Any problem with the UCs propagates through the analysis, design, and code. Therefore, it is essential to expose problems with UCs as early as possible.

We now establish the vocabulary for the rest of the paper: The goal of any RE effort is to elicit and analyze requirements, and eventually, to specify in a *software requirements specification (SRS)* the requirements for the CBS being built. The portion of the real world that a CBS is supposed to automate is the CBS's *domain system (DS)*. During RE analysis for a CBS, the analysts typically develop the *domain system model (DSM)*, which is a model of the CBS's DS. A *UC* of a CBS is one particular way a user of the CBS uses the CBS to achieve a user's goal. The description of a UC is typically given at the shared-interface level, showing the CBS as a monolithic black-box. A popular notation for modeling behavior is *StateCharts (SC)* [8], and among the artifacts we suggest to be included in the SRS for a CBS, are a *UC SC (UCSC)*, a SC representation of each UC of the CBS and a *DS SC (DSSC)*, a SC representation of the CBS's DSM.

Each of the authors has been involved in teaching a course entitled "Software Requirements Specifications" (CS445) [16] at the University of Waterloo for at least six years. The term-long group project for this course is to determine the requirements and to write an SRS for a large VoIP telephony system and its related account-management system. The course teaches UC-driven requirements analysis methods, and the groups are expected to apply them to complete their projects. Svetinovic *et al.* [19] describe the difficulties many of the problems groups were having, which were shown to be independent of the size of the CBS being specified. One of us, Berry, has taught a graduate course entitled "Advanced Topics in Requirements Engineering" (CS846) [2] 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 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.

After first noticing the problems with the SRSs in CS445, we began to search for a way to teach the students to do a better job in their projects. We have been exploring the literature on UC-driven analysis methods [3, 7, 12, e.g.,]. We have been examining different variations of these methods on our own CBS modeling problems. We have been experimenting with advice to give to the students about these methods. With the students' help and feedback, we have slowly iterated to the method that our students have been able to apply and with which the quality of the resulting SRSs has been noticeably improved. The method, which builds on using SCs to model UCs and then unifying the UCSCs into a DSSC [4, 23, 10], is called "SUM (StateChart Unifying Method)".[1] This paper describes the thinking that led to DSSCs and to SUM. The usefulness of DSSCs and the effectiveness of SUM were validated through evaluation of 46 SRSs specified by 149 upper-year software engineering, electrical and computer engineering, and computer science undergraduate and graduate students, each with none to several years of software development experience.

Section 2 reviews the problems with UC-driven requirements analysis methods as we saw it. Section 3 explains our research method and its limitations. Section 4 describes DSSCs, the main artifact produced in SUM. Section 5 describes the good, bad, and ugly effects of carrying out SUM. Section 6 compares our work to related work, and Section 7 concludes the paper.

[1]We name the method only to make it easier to distinguish it from the other methods we mention in this paper. We did not set out to create a new method, and we created a descriptive name for it only during the writing of this paper.

2. The Problem

A typical 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 often results in an SRS that is difficult for both customers and designers to understand. The lack of obvious connections among the different kinds of requirements makes it difficult to determine if the SRS is complete, consistent, and correct. UCs [1, e.g.,] 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 customers. 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 customer and the analyst [1, e.g.,].

Prior to starting our case studies, we hoped that some new artifact based on the UCs would allow analysts to produce even more complete, consistent, and correct SRSs. Perhaps, the same way that UCs help put functional requirements into context, this new artifact based on the UCs would help an analyst to

- detect and fix missing functionality,
- detect and fix functionality across multiple abstraction levels,
- detect and refine inconsistent amounts of detail, i.e., over and under specification,
- discover relationships, e.g., concurrency among UCs and functional requirements, and
- find the *big picture* DSM.

In the typical UC-driven requirements analysis method, UC discovery is followed by drawing sequence diagrams for the UCs and breaking down the CBS's side of the UCs into the CBS's components, to yield a conceptual decomposition of the CBS [12, e.g.,]. This kind of approach was taught to our students for several years. A less common alternative method is to follow UC discovery by writing UCSCs [3, 12, 7]. Nevertheless, in each method, a UC is an artifact at the widest *scope*. Scope refers to the number of functional requirements specified using the artifact. A sequence diagram, a SC, or any other description of a UC is at a scope equal to or less than that of the UC. To arrive at the big picture DSM, it was necessary to proceed in the opposite direction. Rather than decomposing the UCs, as suggested in many UC-driven requirements analysis methods, we discovered that it is better to unify the UCs into a DSM, as suggested in other methods [4, 23, 10].

3. Research Method

Once we decided to use SCs as the notation in which to unify the UCs, what remained was to develop a unification method and to apply it in practice. The method is derived from several sources. Our method is primarily Larman's UC-driven iterative method [12]. The principles of constructing a SC of a DSM are based on Douglass's and Gomaa's principles of UCSC construction [3, 7]. The underlying DSM semantics is based on Glinz's [5, 6].

As mentioned, our method was arrived at iteratively, specifically taking into account what we learned in the first case study described in this paper. In that study, how to perform the unification of UCs into a DSM was left as part of the engineering problem that analysts are supposed to solve. Prior to beginning the first study, we were aware of the related work of Glinz; Whittle and Schumann (W+S); and Harel, Kugler, and Pnueli (H+K+P) [4, 23, 10], but we explicitly decided not to incorporate any of the related work into ours in order to not to constrain the students on the approach and to be able to perform an independent feasibility evaluation. We wanted students to tackle building DSSCs as an engineering problem that they had to solve. In fact, the goal of the first case study was that the students find their own methods. Only after coming to our own conclusions, we compared our results with those of the related work. This comparison is presented in Section 6.

The first case study involved one specification of the Turnstile CBS [18], produced collaboratively by Svetinovic and the students attending tutorial sessions of the CS445 and the CS846 classes. The second set of case studies involved 12 medium-sized specifications of the controller for a two-elevator system in a low-rise building, produced as individual long-term projects in the CS846 class. The third set of case studies involved 34 large-sized specifications of a VoIP system and its account management system, produced as group long-term projects in the CS445 class.

3.1. Threats to Validity

The main focus of this research was not to perform a controlled experiment, but rather to perform case studies under as realistic conditions as possible and to uncover the strengths and weaknesses of a proposed method. To achieve this realism, we did not exercise any of the following usual experimental controls:

- enforcing a particular method upon the subjects,
- enforcing a particular group organization or division of work,
- having each subject work only on the one artifact that we wished to evaluate, and
- limiting the size of the CBS and DS being specified.

With regard to the first missing control, not only did we not enforce any particular method, we simply described the SRS to produce and left the method up to each subject or group of subjects. Clearly, the lack of these four controls yields too much variability for these case studies to be considered a controlled experiment. Thus, the threats to validity of this case study are exactly the same as to any other uncontrolled software engineering study.

Nevertheless, the large number of subjects and the high consistency of the results despite all the variability provides strong support for accepting the usefulness of DSSCs and the potential effectiveness of SUM. The case for the effectiveness of DSSCs is stronger than for the effectiveness of SUM, because we did not force subjects to use SUM, but we did require each team to produce and present a DSSC in their SRS. In fact, the DSSCs proved to be useful no matter how they were produced!

4. Domain System StateCharts

The method that we arrived at, SUM, is based on a very simple idea inspired by observing practice: an effective way to unify a complete set of UCs into a DSM for the CBS is to perform the unification in the SC notation. That is, if each UC in the set can described with a UCSC [3, 7, e.g.,], then it should be possible to unify these UCSCs into a DSSC that describes a high quality DSM [4, 23, 10]. The method depends on the analysts' having specified the UCs' behaviors in UCSCs. However, after practice, an analyst can learn to proceed directly from UCs to a DSSC without having given UCSCs for the UCs. Indeed, we found many a student skipping the production of UCSCs and still producing a good DSSC. Douglass [3] summarizes the advantages of specifying a UC's behavior using a UCSC in a single paragraph:

> *Another means by which use case 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 use case behavior.*

We have recognized an additional advantage of SCs, that of being able to help an analyst to unify a set of UCSCs into a single DSSC. As mentioned, unifying UCs using SCs widens rather than narrows scope. Widening scope leads to exposing problems that might still exist in the individual UCs in the same way that the widening scope during the unification of functional requirements leads to exposing problems that exist in the individual functional requirements.

The rest of this section discusses the semantics of DSSCs and then describes the process of SUM.

4.1. DSSC Semantics

A SC is a *higraph* [9]. It can be used to model just about anything. Thus the first, and most important, step in using SCs is to clearly state what is being modeled. An explicit agreement is needed on what a state represents. There are various definitions of "state". The most common is something like "A state is an ontological condition that persists for a significant period of time." [3] In practice, states are used to capture, for example, any configuration of the object's variables or any activity occurring within the system [3, e.g.,].

For our work, the most appropriate semantics for DSSCs can be described in terms of goals. Note that we defined these semantics *after* the case studies were finished. We *started* with a simple semantics in which a state is either (1) any configuration of variable values or (2) an activity of interest.

Goal-driven RE [14, 21, e.g.,] is a method that 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 business 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 a goal-driven RE method focuses on determining CBS requirements through analysis of *personal* and *business* goals, the method has been used to enhance traditional RE methods, among which are use-case-driven requirements analysis methods [1, e.g.,]. In our case, we started by determining the UCs for a CBS being built by considering the goals for the CBS. It was natural to preserve the goals as part of the DSSC.

Goals capture the *intention* and the *target condition* 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 system example of the previous paragraph, 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 DSSC for a CBS is more general than the traditional state, which is only a configuration of values of CBS variables, and can be very tedious to specify when there are many variables in a CBS. A state in a DSSC can be either (1) an activity of the CBS or (2) a goal that captures the target condition of a part of or of the entire CBS. In the latter case, the goal represents the *postcondition* that describes the impact that the activity in the previous node or on the incoming transition of the DSSC has on the CBS. While the semantics of DSSC nodes is different from that in traditional SC semantics, the semantics of DSSC transitions is consistent with that in traditional SC semantics.

"Why not use UML activity diagrams [15] instead of SCs?" was asked many times because of the presence of states representing activities in our DSSCs. 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 SC, as a Petri net, or as a flowchart [3].
- Laying out and managing a large activity diagram is more difficult, in our experience, than laying out and managing a large SC.
- By definition, it is harder to show, using *activity nodes*, anything but activities in an activity diagram [3]. This also implies 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 that an activity diagram is tailored for, and these are all easier to represent using SCs.

Moreover, we did not find any features provided by activity diagrams that are not provided by SCs.

4.2. Process

The SUM process emerged from the initial case studies and we recommended its steps to the students in the later case studies. The sequential ordering of steps is only for the presentation in this paper. 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.

For the CBS S to be built:

Step 1: Specify UCs for S:

- Identify S's main *business 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 functional subsystems.

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

Step 3: Draw UML *system sequence diagrams* [12] 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: Draw D's DSSC.

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

In addition, we gave the students the following guideline for reducing DSSC rework due to activity refinements: 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 SUM.

5. Evaluation

The first case study was the collaborative production of 3 DSSCs during 3 tutorial sessions of the CS445 and CS846 courses. In each session, the first author, Svetinovic, facilitated the collaboration but tried his best not to influence the solution, which was the responsibility of the students participating in the session. The first DSSC was produced by 12 CS846 students, the second by about 80 CS445 students, and the third by about 50 CS445 students.

The goal of this case study was to practice applying the ideas of Section 4 and to observe building a DSSC for a very small CBS. We chose the Turnstile CBS SRS [20] and used its UCs as the starting point.

The primary value of the first case study was observing (1) three different groups of about 140 students total thoroughly analyzing a small system to produce three DSSCs and (2) the feedback the production of these DSSCs had on the UCs in the earlier, Website-published, and supposedly polished UCs [20]. It was valuable also to see the quality of DSSCs produced by undergraduate students who were novices at SC modeling. The case study showed the amount of improvement in the quality of modeling that can be expected when a lot of people are attacking a small problem, allowing us to estimate the improvement that could be expected when a normal sized workforce attacks a larger, industrial sized problem. A full specification derived from the results of the case study is available at our OODA Website [18].

The second case study yielded 12 SRSs for the controller for a two-elevator system in a low-rise building, a medium-sized CBS. Each SRS was produced by one CS846 graduate student working independently. Each student handed in two preliminary draft SRSs before handing in his final SRS. Each successive draft SRS was required to show a growing set of specific artifacts; in particular, the first draft had to show the owner's complete set of UCs for the CBS. Each student was allowed to see all students' sets of UCs before handing in his second draft so that each student could have as good a set of UCs as possible before constructing his DSSC. However, thereafter, no student was allowed to see any more of each other's work. The complete set of draft and final SRSs can be found at the CS846 Website [2].

The third case study yielded 34 SRSs for a VoIP system and its account management system, a large-sized CBS. Each SRS was produced by a group of three or four primarily undergraduate CS445 students working together; in fact, 4-member groups were in the majority. Each group handed in two preliminary draft SRSs before handing in its final SRS. Each successive draft 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. Each group worked with its own teaching

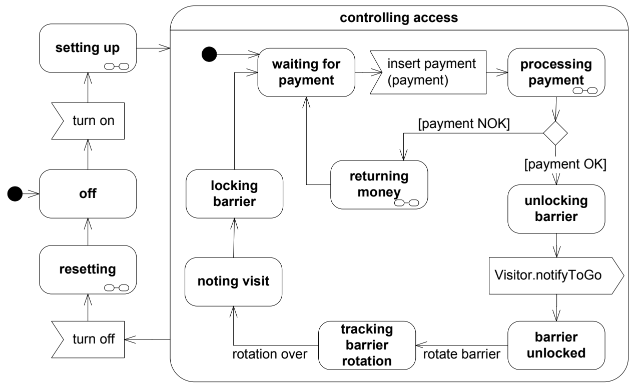


Figure 1. DSSC for the Turnstile CBS

assistant (TA), who served as its customer in a simulated customer–analysts relationship. We cannot make the SRSs available publicly since each offering of the course uses basically the same problem [2], but the SRSs can be provided for independent peer evaluation of our results.

The results of these case studies are mostly good, but there are a few bad and even some ugly results. While positive results were observed in all three case studies, we have chosen to present the examples from the smallest case study in this paper as they require the least explanation. The negative results, which have mostly to do with working with large CBSs, must be explained with examples from the third case study. Thus, we note that the third case study served as a real test for the usefulness of the DSSC and for the effectiveness of SUM.

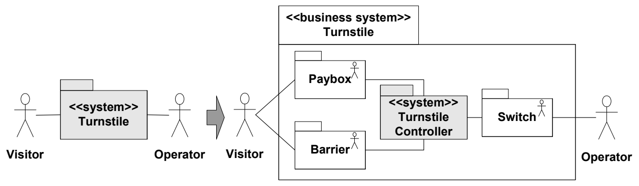
5.1. The Good

Figure 1 shows the final DSSC[2] developed in the first case study from the initial 3 UCs chosen from an earlier SRS for the Turnstile CBS [20]. As mentioned, this DSSC was developed collaboratively in each of three sessions with a total of about 140 students.

The positive effects of building a DSSC by unifying the UCs were apparent not only for the small CBS of the first case study, but also for the larger CBSs for the other case studies. This suggests that the usefulness of building a DSSC from the CBS's UCs does not depend on the size of the CBS.

Each positive result comes from the *process* of unifying the UCs into a DSSC. Furthermore, each positive result is only that *when* the unifying was being done, it was *easier than in the past for an analyst to* do something beneficial. It would not be proper to state the results more strongly. Therefore, the statement of each positive result should begin with "when unifying UCs of a CBS into a DSSC, it was easier than in the past for an analyst to". Since this long phrase would be repeated 7 times, to save some space, we abbreviate it in the following as: "Unification helps to".

[2]The SC syntax conforms to the UML 2.0 standard.



<<business system>> Turnstile
<<system>> Turnstile
Visitor
Operator
Paybox
Barrier
<<system>> Turnstile Controller
Switch

Figure 2. CBS Boundary Change

The first positive result is that Unification helps to **identify the boundary of the CBS.** The set of actors and UCs for a CBS both depends on and helps 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 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 CBS, the boundary established in the SRS [20] from which we took the initial 3 UCs proved to be wrong. The correct boundary is as shown in Figure 2.

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.

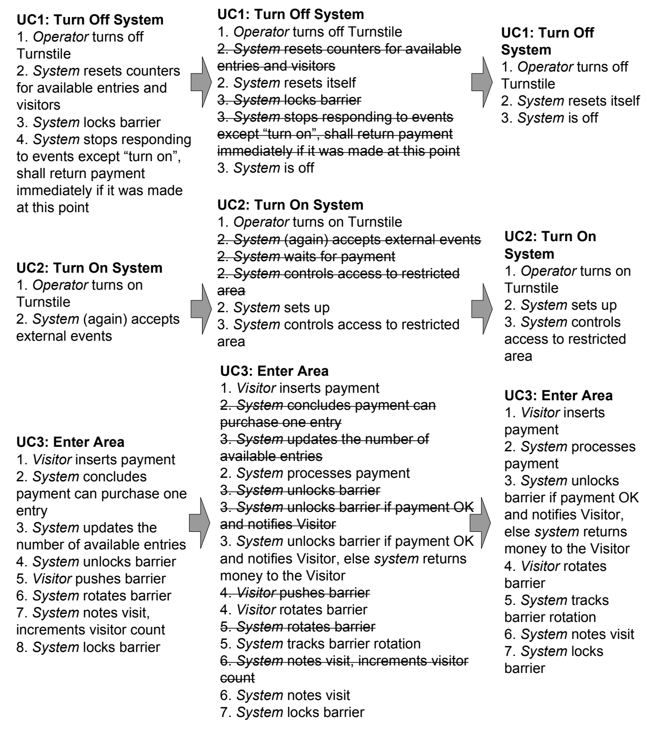
The second positive result is that Unification helps to **identify abstraction level clashes and redundant steps in the UCs.** Correcting the abstraction levels of UCs is necessary to unify successfully the UCs of a CBS into a DSSC. For example, in the original SRS for the Turnstile CBS, the abstraction 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 to write both UCs with no reference to internals. Each decomposition was left to appear in the DSSC.

The third positive result is that Unification helps to **identify incorrect ordering of steps in a UC's description.** It is often the case that a UC's steps 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 CBS, in Step 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 step. Including this exception into the DSSC proved to be awkward, if not impossible, and more analysis was needed to find its rightful place.

The fourth positive result is that 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 of the study into a DSSC did help expose functions of the CBS that were missing in the UCs. The kind of rework appearing in the middle column of Figure 3 is typical. The left column of Figure 3 shows the original UCs of the Turnstile CBS *before* specification of DSSC, the middle column shows modifications to the UCs *during* specification of DSSC, and the right column shows UCs *after* DSSC 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 DSSC is doing, and is doing before coding begins.

The fifth positive result was that Unification helps to **simplify the descriptions of UCs.** This result is a natural follow on of the first four. Building a DSSC almost universally led to simplifying and clarifying the descriptions of the UCs that were being unified. In many cases, a step that was a full paragraph of text was replaced by a single sentence. Clearly defining goals and activities during construction of a DSSC exposed overly complex descriptions of UCs. Simplifying UC descriptions in turn allowed easier identification of goals, activities, inputs, outputs, and other data.

Manifestations of each of these first five results can be seen in the refinement, shown in Figure 3, of the three initial Turnstile CBS 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, Step 3 of the Turn Off System UC is a postcondition rather than the activity it should be. These


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

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* is off

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

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

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

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* 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 3. Turnstile CBS UC Changes

two examples make it clear that (1) one can never be certain about the quality of UCs and that (2) while unifying UCs into a DSSC does help find problems in the UCs, it cannot guarantee finding all of them. Even with a very small CBS such as Turnstile CBS and even with about 300 eyes and half that many brains, 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 DSSC is that it helps the analyst to *see* problems.

The next four positive results were observable only in the two case studies with the larger CBSs.

The sixth positive result was that Unification helps to **see how to restructure the descriptions of the UCs.** We saw that many a student restructured the descriptions of his or her UCs after finishing the DSSC. Typically, the student used pseudocode to describe UC steps, 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.

The seventh positive result was that 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 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 DSSC shows clearly which UCs can be unified temporally, i.e., can be executed concurrently.

The eighth positive result was that **the benefits of unifying UCs of a CBS into a DSSC 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 SUM he or she used. Some wrote UCSCs for the UCs before unifying the UCs into a DSSC, and some unified directly from the original UC descriptions. Regardless of how a student or group did the unification, the resulting DSSCs and the resulting SRSs were significantly better than past experience in the classes led us to expect.

The ninth positive result was that, on average, the CS445 group in the SUM-using term, which had higher evaluation criteria and stricter marking, got a grade for the SRS that was the same as that in previous terms when SUM was not used. Thus, the SUM-assisted SRSs were of higher quality.

In summary, all the positive results and the deepened understanding of the DS can be attributed to the the ability of a DSSC to provide a big picture of the DSM more systematically and more formally than is possible with only UCs. Nevertheless, not all results were positive, as the next two subsections show.

5.2. The Bad

This section discusses pernicious and persistent, i.e., *bad*, problems that remain despite all our best efforts. We hope that they can be resolved by carrying out more studies in the future.

The first bad 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 DSSC. Nevertheless, an occasional student did include in his or her DSSC what was not covered by agreed upon DSSC semantics. Despite the grade penalty for inclusion of non-conventional DSSC 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 bad problem that we observed is the lack of direct support in DSSCs for representation of concepts and objects. SUM is supposed to be a part of an object-oriented domain analysis method for our course projects. As such, conceptual decomposition is supposed to follow the completion of the DSSC. One of the decomposition steps is assigning to concepts the activities captured during unification to the DSSC. 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 his or her DSSC some high-level conceptual decomposition constructs such as subsystems. This tendency suggests the usefulness of extending SC notation with some structural decomposition notation, as suggested by Glinz [6].

The third bad problem that we observed was method related. Many a student claimed that it was easier (1) to grasp all UCs together and then to build the DSSC than (2) to unify UCs one by one into a growing DSSC in either of the two ways suggested by SUM. 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 just jumped directly to producing a DSSC, which turned out to require lots of rework. We say “perceived effort”, because in the end, the thinking needed to fix the poor DSSC was the same as would be needed to redo the UC descriptions. The typical directly produced DSSC had a large number of inconsistencies with use cases, was more difficult to refine, and was at much higher abstraction level than the typical DSSC produced by unifying UCs one by one. Occasionally, the directly produced DSSC 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 now strongly believe that UCs should be unified one by one into a growing DSSC.

The fourth bad problem that we observed was 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 DSSC can require completely redoing all UC descriptions. Many a student, who had written very poor UC descriptions and postponed DSSC specification, simply did not have the time to redo all his or her UC descriptions. To solve this problem, we need give students a clear indication on how much effort it takes to build a DSSC and what the impact of poor UC specifications can be.

5.3. The Ugly

This section describes negative effects that are inherent to the approach and therefore not fixable.

The main negative effect of unifying UCs of a CBS into a DSSC 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 the students in the third Case Study, in the SUM-using term, with that spent by students of previous terms who had specified the same VoIP CBS without any unification.

Teaching students and TAs SUM required 4 hours that were not originally allocated to the course. Of these 4 hours, 2 were spent teaching DSSC unification and 2 were spent teaching how unification fits in the overall RE process. An additional 1 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 SUM 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 terms to about 10 in the SUM-using term. That is, using any variant of SUM required about 25% more elicitation effort. Because we had anticipated at the beginning of the SUM-using term that SUM might require more work, we switched from encouraging 3-person groups to encouraging 4-person groups. In retrospect, the increased specification workload for SUM is proportional to the increase in group size.

The drawback of 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 dealing with *multiple processes and object concurrency*, a difficulty not really addressed by any method. That this difficulty remains with SUM is disappointing because DSSCs are supposed to explicitly expose opportunities for concurrency [3], 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 [6].

6. Related Work

This section compares the work of this paper to that of the three papers whose work appears to be closest, namely papers by Glinz, W+S, and H+K+P [4, 23, 10]. Each of these papers describes one formal treatment of unification of UCs into a SC similar in semantics to our DSSC[3]. Several others, including Somé *et al.* [17], van Lamsweerde *et al.* [22], and Khriss *et al.* [11] have describe algorithms and methods for synthesizing various DSMs, including one in SC notation, from UCs.

Glinz presents a method, intended to be automated, of constructing a SC expression of the DM of a CBS from a set of SCs, 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 SCs must be modified. Glinz's plan was to automate the construction so that analysis, including checking for inconsistencies, can be automated as well.

H+K+P describes an algorithmic method to synthesize a SC expression of a DM 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.

W+S describe an algorithmic method to generate a SC expression of a DM of a CBS from a set of SDs, one for each UC of the CBS. W+S 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 SC expression of the DM that are outside the SDs, e.g., data and preconditions; or both.

There are many analogies between the steps, restrictions, and problems in the methods and algorithms of Glinz, W+S, and H+K+P and those of SUM, not atypical of analogies between other pairs of automated and manual processes. Moreover, the benefits that they observe of their methods and algorithms are consistent with the benefits we observed of SUM. Thus, it can be said that our work and their work constitute independent confirmations of each other.

Our case studies have demonstrated the usefulness and practicality of SUM, a method similar to the UC unification methods described by Glinz, W+S, and H+K+P. Moreover, SUM has been used on CBSs of significant size and has been carried out by a large number of students lacking expertise in SCs and domain modeling. Our studies have shown SUM to provide specific practical benefits to the analysts who apply it and have exposed the drawbacks of the method. Unlike any formal treatment, a case study of actual method use can measure the cost of applying the method. In particular our studies have shown that adding to RE our

[3]All of the works described in this section use the term "scenario" for what we call "UC".

method of unifying UCs of a CBS into a DSSC for the CBS increases the cost of requirements elicitation and the subsequent analysis by about 25%. Because the analysts in our studies were students with no expertise in either SCs or domain modeling, this cost increase is probably 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 our students, having heavily sweated manual unification would greatly appreciate both either of the W+S or the H+K+P tool and the discipline required to prepare the input to the tool.

7. Conclusion

This paper describes the results of case studies that evaluate a practical method for unifying UCs of a CBS into a DSSC for the CBS that can be used as part of RE 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 (1) refined the method and (2) validated the usefulness of the DSSC and the effectiveness of the method both to improve the starting UCs and to yield a quality DSSC that becomes part of a quality SRS.

References

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

[2] CS846 course project. `http://se.uwaterloo.ca/~dberry/ATRE/ElevatorSRSs/`; accessed January 30, 2006.

[3] B. P. Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] M. 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.

[5] M. 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] M. Glinz, S. Berner, and S. Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[7] H. 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.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[9] D. Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988.

[10] D. Harel, H. Kugler, and A. 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, Jan 2005.

[11] I. Khriss, M. Elkoutbi, and R. 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.

[12] C. 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.

[13] S. 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.

[14] J. Mylopoulos, L. Chung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, second edition, 2004.

[16] SE463/CS445 course project. `http://www.student.cs.uwaterloo.ca/~cs445/`; accessed January 30, 2006.

[17] S. Somé, R. Dssouli, and J. 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.

[18] D. Svetinovic. Object-oriented domain analysis (ooda) example: a turnstile system. `http://reqs.org/ooda/examples/ooda-turnstile-example.pdf`; accessed January 30, 2006.

[19] D. Svetinovic, D. M. Berry, and M. 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.

[20] Turnstile system. `http://swag.uwaterloo.ca/~dsvetinovic/turnstilesystem.pdf`; accessed January 30, 2006.

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

[22] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. Softw. Eng.*, 24(12):1089–1114, 1998.

[23] J. Whittle and J. 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.